```
**********************************************************
  174341 Thu Sep  1 02:19:26 2011
new/src/share/vm/opto/memnode.cpp
**********************************************************
______unchanged_portion_omitted_

1485 //-----------------------------Value---------------------------------------
1486 const Type *LoadNode::Value( PhaseTransform *phase ) const {
1487   // Either input is TOP ==> the result is TOP
1488   Node* mem = in(MemNode::Memory);
1489   const Type *t1 = phase->type(mem);
1490   if (t1 == Type::TOP)  return Type::TOP;
1491   Node* adr = in(MemNode::Address);
1492   const TypePtr* tp = phase->type(adr)->isa_ptr();
1493   if (tp == NULL || tp->empty())  return Type::TOP;
1494   int off = tp->offset();
1495   assert(off != Type::OffsetTop, "case covered by TypePtr::empty");
1496   Compile* C = phase->C;
1497 #endif /* ! codereview */

1499   // Try to guess loaded type from pointer type
1500   if (tp->base() == Type::AryPtr) {
1501     const Type *t = tp->is_aryptr()->elem();
1502     // Don't do this for integer types. There is only potential profit if
1503     // the element type t is lower than _type; that is, for int types, if _type
1504     // more restrictive than t.  This only happens here if one is short and the
1505     // char (both 16 bits), and in those cases we've made an intentional decisio
1506     // to use one kind of load over the other. See AndINode::Ideal and 4965907.
1507     // Also, do not try to narrow the type for a LoadKlass, regardless of offset
1508     //
1509     // Yes, it is possible to encounter an expression like (LoadKlass p1:(AddP x
1510     // where the _gvn.type of the AddP is wider than 8.  This occurs when an ear
1511     // copy p0 of (AddP x x 8) has been proven equal to p1, and the p0 has been
1512     // subsumed by p1.  If p1 is on the worklist but has not yet been re-transfo
1513     // it is possible that p1 will have a type like Foo*[int+]:NotNull*+any.
1514     // In fact, that could have been the original type of p1, and p1 could have
1515     // had an original form like p1:(AddP x x (LShiftL quux 3)), where the
1516     // expression (LShiftL quux 3) independently optimized to the constant 8.
1517     if ((t->isa_int() == NULL) && (t->isa_long() == NULL)
1518         && Opcode() != Op_LoadKlass && Opcode() != Op_LoadNKlass) {
1519       // t might actually be lower than _type, if _type is a unique
1520       // concrete subclass of abstract class t.
1521       // Make sure the reference is not into the header, by comparing
1522       // the offset against the offset of the start of the array's data.
1523       // Different array types begin at slightly different offsets (12 vs. 16).
1524       // We choose T_BYTE as an example base type that is least restrictive
1525       // as to alignment, which will therefore produce the smallest
1526       // possible base offset.
1527       const int min_base_off = arrayOopDesc::base_offset_in_bytes(T_BYTE);
1528       if ((uint)off >= (uint)min_base_off) {  // is the offset beyond the header
1529         const Type* jt = t->join(_type);
1530         // In any case, do not allow the join, per se, to empty out the type.
1531         if (jt->empty() && !t->empty()) {
1532           // This can happen if a interface-typed array narrows to a class type.
1533           jt = _type;
1534         }

1536         if (EliminateAutoBox && adr->is_AddP()) {
1537           // The pointers in the autobox arrays are always non-null
1538           Node* base = adr->in(AddPNode::Base);
1539           if (base != NULL &&
1540               !phase->type(base)->higher_equal(TypePtr::NULL_PTR)) {
1541             Compile::AliasType* atp = C->alias_type(base->adr_type());
1496             Compile::AliasType* atp = phase->C->alias_type(base->adr_type());
1542             if (is_autobox_cache(atp)) {
1543               return jt->join(TypePtr::NOTNULL)->is_ptr();
```



```
1544             }
1545           }
1546         }
1547         return jt;
1548       }
1549     }
1550   } else if (tp->base() == Type::InstPtr) {
1551     ciEnv* env = C->env();
1552 #endif /* ! codereview */
1553     const TypeInstPtr* tinst = tp->is_instptr();
1554     ciKlass* klass = tinst->klass();
1555     assert( off != Type::OffsetBot ||
1556             // arrays can be cast to Objects
1557             tp->is_oopptr()->klass()->is_java_lang_Object() ||
1558             // unsafe field access may not have a constant offset
1559             C->has_unsafe_access(),
1506             phase->C->has_unsafe_access(),
1560             "Field accesses must be precise" );
1561     // For oop loads, we expect the _type to be precise
1562     if (klass == env->String_klass() &&
1509     if (klass == phase->C->env()->String_klass() &&
1563         adr->is_AddP() && off != Type::OffsetBot) {
1564       // For constant Strings treat the final fields as compile time constants.
1565       Node* base = adr->in(AddPNode::Base);
1566       const TypeOopPtr* t = phase->type(base)->isa_oopptr();
1567       if (t != NULL && t->singleton()) {
1568         ciField* field = env->String_klass()->get_field_by_offset(off, false);
1515         ciField* field = phase->C->env()->String_klass()->get_field_by_offset(of
1569         if (field != NULL && field->is_final()) {
1570           ciObject* string = t->const_oop();
1571           ciConstant constant = string->as_instance()->field_value(field);
1572           if (constant.basic_type() == T_INT) {
1573             return TypeInt::make(constant.as_int());
1574           } else if (constant.basic_type() == T_ARRAY) {
1575             if (adr->bottom_type()->is_ptr_to_narrowoop()) {
1576               return TypeNarrowOop::make_from_constant(constant.as_object(), tru
1577             } else {
1578               return TypeOopPtr::make_from_constant(constant.as_object(), true);
1579             }
1580           }
1581         }
1582       }
1583     }
1584     // Optimizations for constant objects
1585     ciObject* const_oop = tinst->const_oop();
1586     if (const_oop != NULL) {
1587       // For constant CallSites treat the target field as a compile time constan
1588       if (const_oop->is_call_site()) {
1589         ciCallSite* call_site = const_oop->as_call_site();
1590         ciField* field = call_site->klass()->as_instance_klass()->get_field_by_o
1591         if (field != NULL && field->is_call_site_target()) {
1592           ciMethodHandle* target = call_site->get_target();
1593           if (target != NULL) {  // just in case
1594             ciConstant constant(T_OBJECT, target);
1595             const Type* t;
1596             if (adr->bottom_type()->is_ptr_to_narrowoop()) {
1597               t = TypeNarrowOop::make_from_constant(constant.as_object(), true);
1598             } else {
1599               t = TypeOopPtr::make_from_constant(constant.as_object(), true);
1600             }
1601             // Add a dependence for invalidation of the optimization.
1602             if (!call_site->is_constant_call_site()) {
1603               C->dependencies()->assert_call_site_target_value(call_site, target
1604             }
1605             return t;
1606           }
```

```
1607         }
1608       }
1609     }
1610 #endif /* ! codereview */
1611   } else if (tp->base() == Type::KlassPtr) {
1612     assert( off != Type::OffsetBot ||
1613             // arrays can be cast to Objects
1614             tp->is_klassptr()->klass()->is_java_lang_Object() ||
1615             // also allow array-loading from the primary supertype
1616             // array during subtype checks
1617             Opcode() == Op_LoadKlass,
1618             "Field accesses must be precise" );
1619     // For klass/static loads, we expect the _type to be precise
1620   }
1621
1622   const TypeKlassPtr *tkls = tp->isa_klassptr();
1623   if (tkls != NULL && !StressReflectiveCode) {
1624     ciKlass* klass = tkls->klass();
1625     if (klass->is_loaded() && tkls->klass_is_exact()) {
1626       // We are loading a field from a Klass metaobject whose identity
1627       // is known at compile time (the type is "exact" or "precise").
1628       // Check for fields we know are maintained as constants by the VM.
1629       if (tkls->offset() == Klass::super_check_offset_offset_in_bytes() + (int)s
1630         // The field is Klass::_super_check_offset.  Return its (constant) value
1631         // (Folds up type checking code.)
1632         assert(Opcode() == Op_LoadI, "must load an int from _super_check_offset"
1633         return TypeInt::make(klass->super_check_offset());
1634       }
1635       // Compute index into primary_supers array
1636       juint depth = (tkls->offset() - (Klass::primary_supers_offset_in_bytes() +
1637       // Check for overflowing; use unsigned compare to handle the negative case
1638       if( depth < ciKlass::primary_super_limit() ) {
1639         // The field is an element of Klass::_primary_supers.  Return its (const
1640         // (Folds up type checking code.)
1641         assert(Opcode() == Op_LoadKlass, "must load a klass from _primary_supers
1642         ciKlass *ss = klass->super_of_depth(depth);
1643         return ss ? TypeKlassPtr::make(ss) : TypePtr::NULL_PTR;
1644       }
1645       const Type* aift = load_array_final_field(tkls, klass);
1646       if (aift != NULL)  return aift;
1647       if (tkls->offset() == in_bytes(arrayKlass::component_mirror_offset()) + (i
1648           && klass->is_array_klass()) {
1649         // The field is arrayKlass::_component_mirror.  Return its (constant) va
1650         // (Folds up aClassConstant.getComponentType, common in Arrays.copyOf.)
1651         assert(Opcode() == Op_LoadP, "must load an oop from _component_mirror");
1652         return TypeInstPtr::make(klass->as_array_klass()->component_mirror());
1653       }
1654       if (tkls->offset() == Klass::java_mirror_offset_in_bytes() + (int)sizeof(o
1655         // The field is Klass::_java_mirror.  Return its (constant) value.
1656         // (Folds up the 2nd indirection in anObjConstant.getClass().)
1657         assert(Opcode() == Op_LoadP, "must load an oop from _java_mirror");
1658         return TypeInstPtr::make(klass->java_mirror());
1659       }
1660     }
1661
1662     // We can still check if we are loading from the primary_supers array at a
1663     // shallow enough depth.  Even though the klass is not exact, entries less
1664     // than or equal to its super depth are correct.
1665     if (klass->is_loaded() ) {
1666       ciType *inner = klass->klass();
1667       while( inner->is_obj_array_klass() )
1668         inner = inner->as_obj_array_klass()->base_element_type();
1669       if( inner->is_instance_klass() &&
1670           !inner->as_instance_klass()->flags().is_interface() ) {
1671         // Compute index into primary_supers array
1672         juint depth = (tkls->offset() - (Klass::primary_supers_offset_in_bytes()
```



```
1673         // Check for overflowing; use unsigned compare to handle the negative ca
1674         if( depth < ciKlass::primary_super_limit() &&
1675             depth <= klass->super_depth() ) { // allow self-depth checks to hand
1676           // The field is an element of Klass::_primary_supers.  Return its (con
1677           // (Folds up type checking code.)
1678           assert(Opcode() == Op_LoadKlass, "must load a klass from _primary_supe
1679           ciKlass *ss = klass->super_of_depth(depth);
1680           return ss ? TypeKlassPtr::make(ss) : TypePtr::NULL_PTR;
1681         }
1682       }
1683     }
1684
1685     // If the type is enough to determine that the thing is not an array,
1686     // we can give the layout_helper a positive interval type.
1687     // This will help short-circuit some reflective code.
1688     if (tkls->offset() == Klass::layout_helper_offset_in_bytes() + (int)sizeof(o
1689         && !klass->is_array_klass() // not directly typed as an array
1690         && !klass->is_interface()  // specifically not Serializable & Cloneable
1691         && !klass->is_java_lang_Object()   // not the supertype of all T[]
1692         ) {
1693       // Note:  When interfaces are reliable, we can narrow the interface
1694       // test to (klass != Serializable && klass != Cloneable).
1695       assert(Opcode() == Op_LoadI, "must load an int from _layout_helper");
1696       jint min_size = Klass::instance_layout_helper(oopDesc::header_size(), fals
1697       // The key property of this type is that it folds up tests
1698       // for array-ness, since it proves that the layout_helper is positive.
1699       // Thus, a generic value like the basic object layout helper works fine.
1700       return TypeInt::make(min_size, max_jint, Type::WidenMin);
1701     }
1702   }
1703
1704   // If we are loading from a freshly-allocated object, produce a zero,
1705   // if the load is provably beyond the header of the object.
1706   // (Also allow a variable load from a fresh array to produce zero.)
1707   const TypeOopPtr *tinst = tp->isa_oopptr();
1708   bool is_instance = (tinst != NULL) && tinst->is_known_instance_field();
1709   if (ReduceFieldZeroing || is_instance) {
1710     Node* value = can_see_stored_value(mem,phase);
1711     if (value != NULL && value->is_Con())
1712       return value->bottom_type();
1713   }
1714
1715   if (is_instance) {
1716     // If we have an instance type and our memory input is the
1717     // programs's initial memory state, there is no matching store,
1718     // so just return a zero of the appropriate type
1719     Node *mem = in(MemNode::Memory);
1720     if (mem->is_Parm() && mem->in(0)->is_Start()) {
1721       assert(mem->as_Parm()->_con == TypeFunc::Memory, "must be memory Parm");
1722       return Type::get_zero_type(_type->basic_type());
1723     }
1724   }
1725   return _type;
1726 }
1727
1728 //------------------------------match_edge-------------------------------------
1729 // Do we Match on this edge index or not?  Match only the address.
1730 uint LoadNode::match_edge(uint idx) const {
1731   return idx == MemNode::Address;
1732 }
1733
1734 //--------------------------LoadBNode::Ideal--------------------------------------
1735 //
1736 //  If the previous store is to the same address as this load,
1737 //  and the value stored was larger than a byte, replace this load
1738 //  with the value stored truncated to a byte.  If no truncation is
```

```
1739 //  needed, the replacement is done in LoadNode::Identity().
1740 //
1741 Node *LoadBNode::Ideal(PhaseGVN *phase, bool can_reshape) {
1742   Node* mem = in(MemNode::Memory);
1743   Node* value = can_see_stored_value(mem,phase);
1744   if( value && !phase->type(value)->higher_equal( _type ) ) {
1745     Node *result = phase->transform( new (phase->C, 3) LShiftINode(value, phase-
1746     return new (phase->C, 3) RShiftINode(result, phase->intcon(24));
1747   }
1748   // Identity call will handle the case where truncation is not needed.
1749   return LoadNode::Ideal(phase, can_reshape);
1750 }

1752 //------------------------LoadUBNode::Ideal-------------------------------------
1753 //
1754 //  If the previous store is to the same address as this load,
1755 //  and the value stored was larger than a byte, replace this load
1756 //  with the value stored truncated to a byte.  If no truncation is
1757 //  needed, the replacement is done in LoadNode::Identity().
1758 //
1759 Node* LoadUBNode::Ideal(PhaseGVN* phase, bool can_reshape) {
1760   Node* mem = in(MemNode::Memory);
1761   Node* value = can_see_stored_value(mem, phase);
1762   if (value && !phase->type(value)->higher_equal(_type))
1763     return new (phase->C, 3) AndINode(value, phase->intcon(0xFF));
1764   // Identity call will handle the case where truncation is not needed.
1765   return LoadNode::Ideal(phase, can_reshape);
1766 }

1768 //------------------------LoadUSNode::Ideal-------------------------------------
1769 //
1770 //  If the previous store is to the same address as this load,
1771 //  and the value stored was larger than a char, replace this load
1772 //  with the value stored truncated to a char.  If no truncation is
1773 //  needed, the replacement is done in LoadNode::Identity().
1774 //
1775 Node *LoadUSNode::Ideal(PhaseGVN *phase, bool can_reshape) {
1776   Node* mem = in(MemNode::Memory);
1777   Node* value = can_see_stored_value(mem,phase);
1778   if( value && !phase->type(value)->higher_equal( _type ) )
1779     return new (phase->C, 3) AndINode(value,phase->intcon(0xFFFF));
1780   // Identity call will handle the case where truncation is not needed.
1781   return LoadNode::Ideal(phase, can_reshape);
1782 }

1784 //------------------------LoadSNode::Ideal-------------------------------------
1785 //
1786 //  If the previous store is to the same address as this load,
1787 //  and the value stored was larger than a short, replace this load
1788 //  with the value stored truncated to a short.  If no truncation is
1789 //  needed, the replacement is done in LoadNode::Identity().
1790 //
1791 Node *LoadSNode::Ideal(PhaseGVN *phase, bool can_reshape) {
1792   Node* mem = in(MemNode::Memory);
1793   Node* value = can_see_stored_value(mem,phase);
1794   if( value && !phase->type(value)->higher_equal( _type ) ) {
1795     Node *result = phase->transform( new (phase->C, 3) LShiftINode(value, phase-
1796     return new (phase->C, 3) RShiftINode(result, phase->intcon(16));
1797   }
1798   // Identity call will handle the case where truncation is not needed.
1799   return LoadNode::Ideal(phase, can_reshape);
1800 }

1802 //=============================================================================
1803 //------------------------------LoadKlassNode::make----------------------------
1804 // Polymorphic factory method:
```



```
1805 Node *LoadKlassNode::make( PhaseGVN& gvn, Node *mem, Node *adr, const TypePtr* a
1806   Compile* C = gvn.C;
1807   Node *ctl = NULL;
1808   // sanity check the alias category against the created node type
1809   const TypeOopPtr *adr_type = adr->bottom_type()->isa_oopptr();
1810   assert(adr_type != NULL, "expecting TypeOopPtr");
1811 #ifdef _LP64
1812   if (adr_type->is_ptr_to_narrowoop()) {
1813     Node* load_klass = gvn.transform(new (C, 3) LoadNKlassNode(ctl, mem, adr, at
1814     return new (C, 2) DecodeNNode(load_klass, load_klass->bottom_type()->make_pt
1815   }
1816 #endif
1817   assert(!adr_type->is_ptr_to_narrowoop(), "should have got back a narrow oop");
1818   return new (C, 3) LoadKlassNode(ctl, mem, adr, at, tk);
1819 }

1821 //------------------------------Value------------------------------------------
1822 const Type *LoadKlassNode::Value( PhaseTransform *phase ) const {
1823   return klass_value_common(phase);
1824 }

1826 const Type *LoadNode::klass_value_common( PhaseTransform *phase ) const {
1827   // Either input is TOP ==> the result is TOP
1828   const Type *t1 = phase->type( in(MemNode::Memory) );
1829   if (t1 == Type::TOP)  return Type::TOP;
1830   Node *adr = in(MemNode::Address);
1831   const Type *t2 = phase->type( adr );
1832   if (t2 == Type::TOP)  return Type::TOP;
1833   const TypePtr *tp = t2->is_ptr();
1834   if (TypePtr::above_centerline(tp->ptr()) ||
1835       tp->ptr() == TypePtr::Null)  return Type::TOP;

1837   // Return a more precise klass, if possible
1838   const TypeInstPtr *tinst = tp->isa_instptr();
1839   if (tinst != NULL) {
1840     ciInstanceKlass* ik = tinst->klass()->as_instance_klass();
1841     int offset = tinst->offset();
1842     if (ik == phase->C->env()->Class_klass()
1843         && (offset == java_lang_Class::klass_offset_in_bytes() ||
1844             offset == java_lang_Class::array_klass_offset_in_bytes())) {
1845       // We are loading a special hidden field from a Class mirror object,
1846       // the field which points to the VM's Klass metaobject.
1847       ciType* t = tinst->java_mirror_type();
1848       // java_mirror_type returns non-null for compile-time Class constants.
1849       if (t != NULL) {
1850         // constant oop => constant klass
1851         if (offset == java_lang_Class::array_klass_offset_in_bytes()) {
1852           return TypeKlassPtr::make(ciArrayKlass::make(t));
1853         }
1854         if (!t->is_klass()) {
1855           // a primitive Class (e.g., int.class) has NULL for a klass field
1856           return TypePtr::NULL_PTR;
1857         }
1858         // (Folds up the 1st indirection in aClassConstant.getModifiers().)
1859         return TypeKlassPtr::make(t->as_klass());
1860       }
1861       // non-constant mirror, so we can't tell what's going on
1862     }
1863     if( !ik->is_loaded() )
1864       return _type;             // Bail out if not loaded
1865     if (offset == oopDesc::klass_offset_in_bytes()) {
1866       if (tinst->klass_is_exact()) {
1867         return TypeKlassPtr::make(ik);
1868       }
1869       // See if we can become precise: no subklasses and no interface
1870       // (Note:  We need to support verified interfaces.)
```

```
1871       if (!ik->is_interface() && !ik->has_subklass()) {
1872         //assert(!UseExactTypes, "this code should be useless with exact types")
1873         // Add a dependence; if any subclass added we need to recompile
1874         if (!ik->is_final()) {
1875           // %%% should use stronger assert_unique_concrete_subtype instead
1876           phase->C->dependencies()->assert_leaf_type(ik);
1877         }
1878         // Return precise klass
1879         return TypeKlassPtr::make(ik);
1880       }
1881
1882       // Return root of possible klass
1883       return TypeKlassPtr::make(TypePtr::NotNull, ik, 0/*offset*/);
1884     }
1885   }
1886
1887   // Check for loading klass from an array
1888   const TypeAryPtr *tary = tp->isa_aryptr();
1889   if( tary != NULL ) {
1890     ciKlass *tary_klass = tary->klass();
1891     if (tary_klass != NULL   // can be NULL when at BOTTOM or TOP
1892         && tary->offset() == oopDesc::klass_offset_in_bytes()) {
1893       if (tary->klass_is_exact()) {
1894         return TypeKlassPtr::make(tary_klass);
1895       }
1896       ciArrayKlass *ak = tary->klass()->as_array_klass();
1897       // If the klass is an object array, we defer the question to the
1898       // array component klass.
1899       if( ak->is_obj_array_klass() ) {
1900         assert( ak->is_loaded(), "" );
1901         ciKlass *base_k = ak->as_obj_array_klass()->base_element_klass();
1902         if( base_k->is_loaded() && base_k->is_instance_klass() ) {
1903           ciInstanceKlass* ik = base_k->as_instance_klass();
1904           // See if we can become precise: no subklasses and no interface
1905           if (!ik->is_interface() && !ik->has_subklass()) {
1906             //assert(!UseExactTypes, "this code should be useless with exact typ
1907             // Add a dependence; if any subclass added we need to recompile
1908             if (!ik->is_final()) {
1909               phase->C->dependencies()->assert_leaf_type(ik);
1910             }
1911             // Return precise array klass
1912             return TypeKlassPtr::make(ak);
1913           }
1914         }
1915         return TypeKlassPtr::make(TypePtr::NotNull, ak, 0/*offset*/);
1916       } else {                  // Found a type-array?
1917         //assert(!UseExactTypes, "this code should be useless with exact types")
1918         assert( ak->is_type_array_klass(), "" );
1919         return TypeKlassPtr::make(ak); // These are always precise
1920       }
1921     }
1922   }
1923
1924   // Check for loading klass from an array klass
1925   const TypeKlassPtr *tkls = tp->isa_klassptr();
1926   if (tkls != NULL && !StressReflectiveCode) {
1927     ciKlass* klass = tkls->klass();
1928     if( !klass->is_loaded() )
1929       return _type;             // Bail out if not loaded
1930     if( klass->is_obj_array_klass() &&
1931         (uint)tkls->offset() == objArrayKlass::element_klass_offset_in_bytes() +
1932       ciKlass* elem = klass->as_obj_array_klass()->element_klass();
1933       // // Always returning precise element type is incorrect,
1934       // // e.g., element type could be object and array may contain strings
1935       // return TypeKlassPtr::make(TypePtr::Constant, elem, 0);
```



```
1937       // The array's TypeKlassPtr was declared 'precise' or 'not precise'
1938       // according to the element type's subclassing.
1939       return TypeKlassPtr::make(tkls->ptr(), elem, 0/*offset*/);
1940     }
1941     if( klass->is_instance_klass() && tkls->klass_is_exact() &&
1942         (uint)tkls->offset() == Klass::super_offset_in_bytes() + sizeof(oopDesc)
1943       ciKlass* sup = klass->as_instance_klass()->super();
1944       // The field is Klass::_super.  Return its (constant) value.
1945       // (Folds up the 2nd indirection in aClassConstant.getSuperClass().)
1946       return sup ? TypeKlassPtr::make(sup) : TypePtr::NULL_PTR;
1947     }
1948   }
1949
1950   // Bailout case
1951   return LoadNode::Value(phase);
1952 }
1953
1954 //------------------------------Identity---------------------------------------
1955 // To clean up reflective code, simplify k.java_mirror.as_klass to plain k.
1956 // Also feed through the klass in Allocate(...klass...)._klass.
1957 Node* LoadKlassNode::Identity( PhaseTransform *phase ) {
1958   return klass_identity_common(phase);
1959 }
1960
1961 Node* LoadNode::klass_identity_common(PhaseTransform *phase ) {
1962   Node* x = LoadNode::Identity(phase);
1963   if (x != this)  return x;
1964
1965   // Take apart the address into an oop and and offset.
1966   // Return 'this' if we cannot.
1967   Node*    adr    = in(MemNode::Address);
1968   intptr_t offset = 0;
1969   Node*    base   = AddPNode::Ideal_base_and_offset(adr, phase, offset);
1970   if (base == NULL)     return this;
1971   const TypeOopPtr* toop = phase->type(adr)->isa_oopptr();
1972   if (toop == NULL)     return this;
1973
1974   // We can fetch the klass directly through an AllocateNode.
1975   // This works even if the klass is not constant (clone or newArray).
1976   if (offset == oopDesc::klass_offset_in_bytes()) {
1977     Node* allocated_klass = AllocateNode::Ideal_klass(base, phase);
1978     if (allocated_klass != NULL) {
1979       return allocated_klass;
1980     }
1981   }
1982
1983   // Simplify k.java_mirror.as_klass to plain k, where k is a klassOop.
1984   // Simplify ak.component_mirror.array_klass to plain ak, ak an arrayKlass.
1985   // See inline_native_Class_query for occurrences of these patterns.
1986   // Java Example:  x.getClass().isAssignableFrom(y)
1987   // Java Example:  Array.newInstance(x.getClass().getComponentType(), n)
1988   //
1989   // This improves reflective code, often making the Class
1990   // mirror go completely dead.  (Current exception:  Class
1991   // mirrors may appear in debug info, but we could clean them out by
1992   // introducing a new debug info operator for klassOop.java_mirror).
1993   if (toop->isa_instptr() && toop->klass() == phase->C->env()->Class_klass()
1994       && (offset == java_lang_Class::klass_offset_in_bytes() ||
1995           offset == java_lang_Class::array_klass_offset_in_bytes())) {
1996     // We are loading a special hidden field from a Class mirror,
1997     // the field which points to its Klass or arrayKlass metaobject.
1998     if (base->is_Load()) {
1999       Node* adr2 = base->in(MemNode::Address);
2000       const TypeKlassPtr* tkls = phase->type(adr2)->isa_klassptr();
2001       if (tkls != NULL && !tkls->empty()
2002           && (tkls->klass()->is_instance_klass() ||
```

```
2003               tkls->klass()->is_array_klass())
2004           && adr2->is_AddP()
2005           ) {
2006         int mirror_field = Klass::java_mirror_offset_in_bytes();
2007         if (offset == java_lang_Class::array_klass_offset_in_bytes()) {
2008           mirror_field = in_bytes(arrayKlass::component_mirror_offset());
2009         }
2010         if (tkls->offset() == mirror_field + (int)sizeof(oopDesc)) {
2011           return adr2->in(AddPNode::Base);
2012         }
2013       }
2014     }
2015   }

2017   return this;
2018 }


2021 //------------------------------Value-----------------------------------------
2022 const Type *LoadNKlassNode::Value( PhaseTransform *phase ) const {
2023   const Type *t = klass_value_common(phase);
2024   if (t == Type::TOP)
2025     return t;

2027   return t->make_narrowoop();
2028 }

2030 //------------------------------Identity---------------------------------------
2031 // To clean up reflective code, simplify k.java_mirror.as_klass to narrow k.
2032 // Also feed through the klass in Allocate(...klass...)._klass.
2033 Node* LoadNKlassNode::Identity( PhaseTransform *phase ) {
2034   Node *x = klass_identity_common(phase);

2036   const Type *t = phase->type( x );
2037   if( t == Type::TOP ) return x;
2038   if( t->isa_narrowoop()) return x;

2040   return phase->transform(new (phase->C, 2) EncodePNode(x, t->make_narrowoop()));
2041 }

2043 //------------------------------Value------------------------------------------
2044 const Type *LoadRangeNode::Value( PhaseTransform *phase ) const {
2045   // Either input is TOP ==> the result is TOP
2046   const Type *t1 = phase->type( in(MemNode::Memory) );
2047   if( t1 == Type::TOP ) return Type::TOP;
2048   Node *adr = in(MemNode::Address);
2049   const Type *t2 = phase->type( adr );
2050   if( t2 == Type::TOP ) return Type::TOP;
2051   const TypePtr *tp = t2->is_ptr();
2052   if (TypePtr::above_centerline(tp->ptr()))  return Type::TOP;
2053   const TypeAryPtr *tap = tp->isa_aryptr();
2054   if( !tap ) return _type;
2055   return tap->size();
2056 }

2058 //-------------------------------Ideal-----------------------------------------
2059 // Feed through the length in AllocateArray(...length...)._length.
2060 Node *LoadRangeNode::Ideal(PhaseGVN *phase, bool can_reshape) {
2061   Node* p = MemNode::Ideal_common(phase, can_reshape);
2062   if (p)  return (p == NodeSentinel) ? NULL : p;

2064   // Take apart the address into an oop and and offset.
2065   // Return 'this' if we cannot.
2066   Node*    adr    = in(MemNode::Address);
2067   intptr_t offset = 0;
2068   Node*    base   = AddPNode::Ideal_base_and_offset(adr, phase,  offset);
```



```
2069   if (base == NULL)     return NULL;
2070   const TypeAryPtr* tary = phase->type(adr)->isa_aryptr();
2071   if (tary == NULL)     return NULL;

2073   // We can fetch the length directly through an AllocateArrayNode.
2074   // This works even if the length is not constant (clone or newArray).
2075   if (offset == arrayOopDesc::length_offset_in_bytes()) {
2076     AllocateArrayNode* alloc = AllocateArrayNode::Ideal_array_allocation(base, p
2077     if (alloc != NULL) {
2078       Node* allocated_length = alloc->Ideal_length();
2079       Node* len = alloc->make_ideal_length(tary, phase);
2080       if (allocated_length != len) {
2081         // New CastII improves on this.
2082         return len;
2083       }
2084     }
2085   }

2087   return NULL;
2088 }

2090 //------------------------------Identity---------------------------------------
2091 // Feed through the length in AllocateArray(...length...)._length.
2092 Node* LoadRangeNode::Identity( PhaseTransform *phase ) {
2093   Node* x = LoadINode::Identity(phase);
2094   if (x != this)  return x;

2096   // Take apart the address into an oop and and offset.
2097   // Return 'this' if we cannot.
2098   Node*    adr    = in(MemNode::Address);
2099   intptr_t offset = 0;
2100   Node*    base   = AddPNode::Ideal_base_and_offset(adr, phase, offset);
2101   if (base == NULL)     return this;
2102   const TypeAryPtr* tary = phase->type(adr)->isa_aryptr();
2103   if (tary == NULL)     return this;

2105   // We can fetch the length directly through an AllocateArrayNode.
2106   // This works even if the length is not constant (clone or newArray).
2107   if (offset == arrayOopDesc::length_offset_in_bytes()) {
2108     AllocateArrayNode* alloc = AllocateArrayNode::Ideal_array_allocation(base, p
2109     if (alloc != NULL) {
2110       Node* allocated_length = alloc->Ideal_length();
2111       // Do not allow make_ideal_length to allocate a CastII node.
2112       Node* len = alloc->make_ideal_length(tary, phase, false);
2113       if (allocated_length == len) {
2114         // Return allocated_length only if it would not be improved by a CastII.
2115         return allocated_length;
2116       }
2117     }
2118   }

2120   return this;

2122 }

2124 //=============================================================================
2125 //---------------------------StoreNode::make-----------------------------------
2126 // Polymorphic factory method:
2127 StoreNode* StoreNode::make( PhaseGVN& gvn, Node* ctl, Node* mem, Node* adr, cons
2128   Compile* C = gvn.C;
2129   assert( C->get_alias_index(adr_type) != Compile::AliasIdxRaw ||
2130           ctl != NULL, "raw memory operations should have control edge");

2132   switch (bt) {
2133   case T_BOOLEAN:
2134   case T_BYTE:    return new (C, 4) StoreBNode(ctl, mem, adr, adr_type, val);
```

```
2135   case T_INT:     return new (C, 4) StoreINode(ctl, mem, adr, adr_type, val);
2136   case T_CHAR:
2137   case T_SHORT:   return new (C, 4) StoreCNode(ctl, mem, adr, adr_type, val);
2138   case T_LONG:    return new (C, 4) StoreLNode(ctl, mem, adr, adr_type, val);
2139   case T_FLOAT:   return new (C, 4) StoreFNode(ctl, mem, adr, adr_type, val);
2140   case T_DOUBLE:  return new (C, 4) StoreDNode(ctl, mem, adr, adr_type, val);
2141   case T_ADDRESS:
2142   case T_OBJECT:
2143 #ifdef _LP64
2144     if (adr->bottom_type()->is_ptr_to_narrowoop() ||
2145         (UseCompressedOops && val->bottom_type()->isa_klassptr() &&
2146          adr->bottom_type()->isa_rawptr())) {
2147       val = gvn.transform(new (C, 2) EncodePNode(val, val->bottom_type()->make_n
2148       return new (C, 4) StoreNNode(ctl, mem, adr, adr_type, val);
2149     } else
2150 #endif
2151     {
2152       return new (C, 4) StorePNode(ctl, mem, adr, adr_type, val);
2153     }
2154   }
2155   ShouldNotReachHere();
2156   return (StoreNode*)NULL;
2157 }

2159 StoreLNode* StoreLNode::make_atomic(Compile *C, Node* ctl, Node* mem, Node* adr,
2160   bool require_atomic = true;
2161   return new (C, 4) StoreLNode(ctl, mem, adr, adr_type, val, require_atomic);
2162 }


2165 //------------------------bottom_type----------------------------------------
2166 const Type *StoreNode::bottom_type() const {
2167   return Type::MEMORY;
2168 }

2170 //------------------------------hash-----------------------------------------
2171 uint StoreNode::hash() const {
2172   // unroll addition of interesting fields
2173   //return (uintptr_t)in(Control) + (uintptr_t)in(Memory) + (uintptr_t)in(Addres

2175   // Since they are not commoned, do not hash them:
2176   return NO_HASH;
2177 }

2179 //------------------------------Ideal------------------------------------------
2180 // Change back-to-back Store(, p, x) -> Store(m, p, y) to Store(m, p, x).
2181 // When a store immediately follows a relevant allocation/initialization,
2182 // try to capture it into the initialization, or hoist it above.
2183 Node *StoreNode::Ideal(PhaseGVN *phase, bool can_reshape) {
2184   Node* p = MemNode::Ideal_common(phase, can_reshape);
2185   if (p)  return (p == NodeSentinel) ? NULL : p;

2187   Node* mem     = in(MemNode::Memory);
2188   Node* address = in(MemNode::Address);

2190   // Back-to-back stores to same address?  Fold em up.  Generally
2191   // unsafe if I have intervening uses...  Also disallowed for StoreCM
2192   // since they must follow each StoreP operation.  Redundant StoreCMs
2193   // are eliminated just before matching in final_graph_reshape.
2194   if (mem->is_Store() && phase->eqv_uncast(mem->in(MemNode::Address), address) &
2195       mem->Opcode() != Op_StoreCM) {
2196     // Looking at a dead closed cycle of memory?
2197     assert(mem != mem->in(MemNode::Memory), "dead loop in StoreNode::Ideal");

2199     assert(Opcode() == mem->Opcode() ||
2200            phase->C->get_alias_index(adr_type()) == Compile::AliasIdxRaw,
```



```
2201            "no mismatched stores, except on raw memory");

2203     if (mem->outcnt() == 1 &&           // check for intervening uses
2204         mem->as_Store()->memory_size() <= this->memory_size()) {
2205       // If anybody other than 'this' uses 'mem', we cannot fold 'mem' away.
2206       // For example, 'mem' might be the final state at a conditional return.
2207       // Or, 'mem' might be used by some node which is live at the same time
2208       // 'this' is live, which might be unschedulable.  So, require exactly
2209       // ONE user, the 'this' store, until such time as we clone 'mem' for
2210       // each of 'mem's uses (thus making the exactly-1-user-rule hold true).
2211       if (can_reshape) {  // (%%% is this an anachronism?)
2212         set_req_X(MemNode::Memory, mem->in(MemNode::Memory),
2213                   phase->is_IterGVN());
2214       } else {
2215         // It's OK to do this in the parser, since DU info is always accurate,
2216         // and the parser always refers to nodes via SafePointNode maps.
2217         set_req(MemNode::Memory, mem->in(MemNode::Memory));
2218       }
2219       return this;
2220     }
2221   }

2223   // Capture an unaliased, unconditional, simple store into an initializer.
2224   // Or, if it is independent of the allocation, hoist it above the allocation.
2225   if (ReduceFieldZeroing && /*can_reshape &&*/
2226       mem->is_Proj() && mem->in(0)->is_Initialize()) {
2227     InitializeNode* init = mem->in(0)->as_Initialize();
2228     intptr_t offset = init->can_capture_store(this, phase);
2229     if (offset > 0) {
2230       Node* moved = init->capture_store(this, offset, phase);
2231       // If the InitializeNode captured me, it made a raw copy of me,
2232       // and I need to disappear.
2233       if (moved != NULL) {
2234         // %%% hack to ensure that Ideal returns a new node:
2235         mem = MergeMemNode::make(phase->C, mem);
2236         return mem;             // fold me away
2237       }
2238     }
2239   }

2241   return NULL;                  // No further progress
2242 }

2244 //------------------------------Value-----------------------------------------
2245 const Type *StoreNode::Value( PhaseTransform *phase ) const {
2246   // Either input is TOP ==> the result is TOP
2247   const Type *t1 = phase->type( in(MemNode::Memory) );
2248   if( t1 == Type::TOP ) return Type::TOP;
2249   const Type *t2 = phase->type( in(MemNode::Address) );
2250   if( t2 == Type::TOP ) return Type::TOP;
2251   const Type *t3 = phase->type( in(MemNode::ValueIn) );
2252   if( t3 == Type::TOP ) return Type::TOP;
2253   return Type::MEMORY;
2254 }

2256 //------------------------------Identity---------------------------------------
2257 // Remove redundant stores:
2258 //   Store(m, p, Load(m, p)) changes to m.
2259 //   Store(, p, x) -> Store(m, p, x) changes to Store(m, p, x).
2260 Node *StoreNode::Identity( PhaseTransform *phase ) {
2261   Node* mem = in(MemNode::Memory);
2262   Node* adr = in(MemNode::Address);
2263   Node* val = in(MemNode::ValueIn);

2265   // Load then Store?  Then the Store is useless
2266   if (val->is_Load() &&
```

```
2267       phase->eqv_uncast( val->in(MemNode::Address), adr ) &&
2268       phase->eqv_uncast( val->in(MemNode::Memory ), mem ) &&
2269       val->as_Load()->store_Opcode() == Opcode()) {
2270     return mem;
2271   }

2273   // Two stores in a row of the same value?
2274   if (mem->is_Store() &&
2275       phase->eqv_uncast( mem->in(MemNode::Address), adr ) &&
2276       phase->eqv_uncast( mem->in(MemNode::ValueIn), val ) &&
2277       mem->Opcode() == Opcode()) {
2278     return mem;
2279   }

2281   // Store of zero anywhere into a freshly-allocated object?
2282   // Then the store is useless.
2283   // (It must already have been captured by the InitializeNode.)
2284   if (ReduceFieldZeroing && phase->type(val)->is_zero_type()) {
2285     // a newly allocated object is already all-zeroes everywhere
2286     if (mem->is_Proj() && mem->in(0)->is_Allocate()) {
2287       return mem;
2288     }

2290     // the store may also apply to zero-bits in an earlier object
2291     Node* prev_mem = find_previous_store(phase);
2292     // Steps (a), (b):  Walk past independent stores to find an exact match.
2293     if (prev_mem != NULL) {
2294       Node* prev_val = can_see_stored_value(prev_mem, phase);
2295       if (prev_val != NULL && phase->eqv(prev_val, val)) {
2296         // prev_val and val might differ by a cast; it would be good
2297         // to keep the more informative of the two.
2298         return mem;
2299       }
2300     }
2301   }

2303   return this;
2304 }

2306 //------------------------------match_edge-------------------------------------
2307 // Do we Match on this edge index or not?  Match only memory & value
2308 uint StoreNode::match_edge(uint idx) const {
2309   return idx == MemNode::Address || idx == MemNode::ValueIn;
2310 }

2312 //------------------------------cmp--------------------------------------------
2313 // Do not common stores up together.  They generally have to be split
2314 // back up anyways, so do not bother.
2315 uint StoreNode::cmp( const Node &n ) const {
2316   return (&n == this);          // Always fail except on self
2317 }

2319 //------------------------------Ideal_masked_input-----------------------------
2320 // Check for a useless mask before a partial-word store
2321 // (StoreB ... (AndI valIn conIa) )
2322 // If (conIa & mask == mask) this simplifies to
2323 // (StoreB ... (valIn) )
2324 Node *StoreNode::Ideal_masked_input(PhaseGVN *phase, uint mask) {
2325   Node *val = in(MemNode::ValueIn);
2326   if( val->Opcode() == Op_AndI ) {
2327     const TypeInt *t = phase->type( val->in(2) )->isa_int();
2328     if( t && t->is_con() && (t->get_con() & mask) == mask ) {
2329       set_req(MemNode::ValueIn, val->in(1));
2330       return this;
2331     }
2332   }
```



```
2333   return NULL;
2334 }


2337 //------------------------------Ideal_sign_extended_input----------------------
2338 // Check for useless sign-extension before a partial-word store
2339 // (StoreB ... (RShiftI _ (LShiftI _ valIn conIL ) conIR) )
2340 // If (conIL == conIR && conIR <= num_bits)  this simplifies to
2341 // (StoreB ... (valIn) )
2342 Node *StoreNode::Ideal_sign_extended_input(PhaseGVN *phase, int num_bits) {
2343   Node *val = in(MemNode::ValueIn);
2344   if( val->Opcode() == Op_RShiftI ) {
2345     const TypeInt *t = phase->type( val->in(2) )->isa_int();
2346     if( t && t->is_con() && (t->get_con() <= num_bits) ) {
2347       Node *shl = val->in(1);
2348       if( shl->Opcode() == Op_LShiftI ) {
2349         const TypeInt *t2 = phase->type( shl->in(2) )->isa_int();
2350         if( t2 && t2->is_con() && (t2->get_con() == t->get_con()) ) {
2351           set_req(MemNode::ValueIn, shl->in(1));
2352           return this;
2353         }
2354       }
2355     }
2356   }
2357   return NULL;
2358 }

2360 //------------------------------value_never_loaded-----------------------------
2361 // Determine whether there are any possible loads of the value stored.
2362 // For simplicity, we actually check if there are any loads from the
2363 // address stored to, not just for loads of the value stored by this node.
2364 //
2365 bool StoreNode::value_never_loaded( PhaseTransform *phase) const {
2366   Node *adr = in(Address);
2367   const TypeOopPtr *adr_oop = phase->type(adr)->isa_oopptr();
2368   if (adr_oop == NULL)
2369     return false;
2370   if (!adr_oop->is_known_instance_field())
2371     return false; // if not a distinct instance, there may be aliases of the add
2372   for (DUIterator_Fast imax, i = adr->fast_outs(imax); i < imax; i++) {
2373     Node *use = adr->fast_out(i);
2374     int opc = use->Opcode();
2375     if (use->is_Load() || use->is_LoadStore()) {
2376       return false;
2377     }
2378   }
2379   return true;
2380 }

2382 //=============================================================================
2383 //------------------------------Ideal------------------------------------------
2384 // If the store is from an AND mask that leaves the low bits untouched, then
2385 // we can skip the AND operation.  If the store is from a sign-extension
2386 // (a left shift, then right shift) we can skip both.
2387 Node *StoreBNode::Ideal(PhaseGVN *phase, bool can_reshape){
2388   Node *progress = StoreNode::Ideal_masked_input(phase, 0xFF);
2389   if( progress != NULL ) return progress;

2391   progress = StoreNode::Ideal_sign_extended_input(phase, 24);
2392   if( progress != NULL ) return progress;

2394   // Finally check the default case
2395   return StoreNode::Ideal(phase, can_reshape);
2396 }

2398 //=============================================================================
```

```
2399 //-----------------------------Ideal---------------------------------------
2400 // If the store is from an AND mask that leaves the low bits untouched, then
2401 // we can skip the AND operation
2402 Node *StoreCNode::Ideal(PhaseGVN *phase, bool can_reshape){
2403   Node *progress = StoreNode::Ideal_masked_input(phase, 0xFFFF);
2404   if( progress != NULL ) return progress;

2406   progress = StoreNode::Ideal_sign_extended_input(phase, 16);
2407   if( progress != NULL ) return progress;

2409   // Finally check the default case
2410   return StoreNode::Ideal(phase, can_reshape);
2411 }

2413 //=============================================================================
2414 //------------------------------Identity---------------------------------------
2415 Node *StoreCMNode::Identity( PhaseTransform *phase ) {
2416   // No need to card mark when storing a null ptr
2417   Node* my_store = in(MemNode::OopStore);
2418   if (my_store->is_Store()) {
2419     const Type *t1 = phase->type( my_store->in(MemNode::ValueIn) );
2420     if( t1 == TypePtr::NULL_PTR ) {
2421       return in(MemNode::Memory);
2422     }
2423   }
2424   return this;
2425 }

2427 //=============================================================================
2428 //------------------------------Ideal------------------------------------------
2429 Node *StoreCMNode::Ideal(PhaseGVN *phase, bool can_reshape){
2430   Node* progress = StoreNode::Ideal(phase, can_reshape);
2431   if (progress != NULL) return progress;

2433   Node* my_store = in(MemNode::OopStore);
2434   if (my_store->is_MergeMem()) {
2435     Node* mem = my_store->as_MergeMem()->memory_at(oop_alias_idx());
2436     set_req(MemNode::OopStore, mem);
2437     return this;
2438   }

2440   return NULL;
2441 }

2443 //------------------------------Value-----------------------------------------
2444 const Type *StoreCMNode::Value( PhaseTransform *phase ) const {
2445   // Either input is TOP ==> the result is TOP
2446   const Type *t = phase->type( in(MemNode::Memory) );
2447   if( t == Type::TOP ) return Type::TOP;
2448   t = phase->type( in(MemNode::Address) );
2449   if( t == Type::TOP ) return Type::TOP;
2450   t = phase->type( in(MemNode::ValueIn) );
2451   if( t == Type::TOP ) return Type::TOP;
2452   // If extra input is TOP ==> the result is TOP
2453   t = phase->type( in(MemNode::OopStore) );
2454   if( t == Type::TOP ) return Type::TOP;

2456   return StoreNode::Value( phase );
2457 }


2460 //=============================================================================
2461 //----------------------------------SCMemProjNode------------------------------
2462 const Type * SCMemProjNode::Value( PhaseTransform *phase ) const
2463 {
2464   return bottom_type();
```



```
2465 }

2467 //=============================================================================
2468 LoadStoreNode::LoadStoreNode( Node *c, Node *mem, Node *adr, Node *val, Node *ex
2469   init_req(MemNode::Control, c  );
2470   init_req(MemNode::Memory , mem);
2471   init_req(MemNode::Address, adr);
2472   init_req(MemNode::ValueIn, val);
2473   init_req(         ExpectedIn, ex );
2474   init_class_id(Class_LoadStore);

2476 }

2478 //=============================================================================
2479 //-----------------------------adr_type--------------------------------------
2480 // Do we Match on this edge index or not?  Do not match memory
2481 const TypePtr* ClearArrayNode::adr_type() const {
2482   Node *adr = in(3);
2483   return MemNode::calculate_adr_type(adr->bottom_type());
2484 }

2486 //------------------------------match_edge-------------------------------------
2487 // Do we Match on this edge index or not?  Do not match memory
2488 uint ClearArrayNode::match_edge(uint idx) const {
2489   return idx > 1;
2490 }

2492 //------------------------------Identity---------------------------------------
2493 // Clearing a zero length array does nothing
2494 Node *ClearArrayNode::Identity( PhaseTransform *phase ) {
2495   return phase->type(in(2))->higher_equal(TypeX::ZERO)  ? in(1) : this;
2496 }

2498 //------------------------------Idealize---------------------------------------
2499 // Clearing a short array is faster with stores
2500 Node *ClearArrayNode::Ideal(PhaseGVN *phase, bool can_reshape){
2501   const int unit = BytesPerLong;
2502   const TypeX* t = phase->type(in(2))->isa_intptr_t();
2503   if (!t)  return NULL;
2504   if (!t->is_con())  return NULL;
2505   intptr_t raw_count = t->get_con();
2506   intptr_t size = raw_count;
2507   if (!Matcher::init_array_count_is_in_bytes) size *= unit;
2508   // Clearing nothing uses the Identity call.
2509   // Negative clears are possible on dead ClearArrays
2510   // (see jck test stmt114.stmt11402.val).
2511   if (size <= 0 || size % unit != 0)  return NULL;
2512   intptr_t count = size / unit;
2513   // Length too long; use fast hardware clear
2514   if (size > Matcher::init_array_short_size)  return NULL;
2515   Node *mem = in(1);
2516   if( phase->type(mem)==Type::TOP ) return NULL;
2517   Node *adr = in(3);
2518   const Type* at = phase->type(adr);
2519   if( at==Type::TOP ) return NULL;
2520   const TypePtr* atp = at->isa_ptr();
2521   // adjust atp to be the correct array element address type
2522   if (atp == NULL)  atp = TypePtr::BOTTOM;
2523   else              atp = atp->add_offset(Type::OffsetBot);
2524   // Get base for derived pointer purposes
2525   if( adr->Opcode() != Op_AddP ) Unimplemented();
2526   Node *base = adr->in(1);

2528   Node *zero = phase->makecon(TypeLong::ZERO);
2529   Node *off  = phase->MakeConX(BytesPerLong);
2530   mem = new (phase->C, 4) StoreLNode(in(0),mem,adr,atp,zero);
```

```
2531   count--;
2532   while( count-- ) {
2533     mem = phase->transform(mem);
2534     adr = phase->transform(new (phase->C, 4) AddPNode(base,adr,off));
2535     mem = new (phase->C, 4) StoreLNode(in(0),mem,adr,atp,zero);
2536   }
2537   return mem;
2538 }

2540 //----------------------------step_through----------------------------------
2541 // Return allocation input memory edge if it is different instance
2542 // or itself if it is the one we are looking for.
2543 bool ClearArrayNode::step_through(Node** np, uint instance_id, PhaseTransform* p
2544   Node* n = *np;
2545   assert(n->is_ClearArray(), "sanity");
2546   intptr_t offset;
2547   AllocateNode* alloc = AllocateNode::Ideal_allocation(n->in(3), phase, offset);
2548   // This method is called only before Allocate nodes are expanded during
2549   // macro nodes expansion. Before that ClearArray nodes are only generated
2550   // in LibraryCallKit::generate_arraycopy() which follows allocations.
2551   assert(alloc != NULL, "should have allocation");
2552   if (alloc->_idx == instance_id) {
2553     // Can not bypass initialization of the instance we are looking for.
2554     return false;
2555   }
2556   // Otherwise skip it.
2557   InitializeNode* init = alloc->initialization();
2558   if (init != NULL)
2559     *np = init->in(TypeFunc::Memory);
2560   else
2561     *np = alloc->in(TypeFunc::Memory);
2562   return true;
2563 }

2565 //----------------------------clear_memory-------------------------------------
2566 // Generate code to initialize object storage to zero.
2567 Node* ClearArrayNode::clear_memory(Node* ctl, Node* mem, Node* dest,
2568                                    intptr_t start_offset,
2569                                    Node* end_offset,
2570                                    PhaseGVN* phase) {
2571   Compile* C = phase->C;
2572   intptr_t offset = start_offset;

2574   int unit = BytesPerLong;
2575   if ((offset % unit) != 0) {
2576     Node* adr = new (C, 4) AddPNode(dest, dest, phase->MakeConX(offset));
2577     adr = phase->transform(adr);
2578     const TypePtr* atp = TypeRawPtr::BOTTOM;
2579     mem = StoreNode::make(*phase, ctl, mem, adr, atp, phase->zerocon(T_INT), T_I
2580     mem = phase->transform(mem);
2581     offset += BytesPerInt;
2582   }
2583   assert((offset % unit) == 0, "");

2585   // Initialize the remaining stuff, if any, with a ClearArray.
2586   return clear_memory(ctl, mem, dest, phase->MakeConX(offset), end_offset, phase
2587 }

2589 Node* ClearArrayNode::clear_memory(Node* ctl, Node* mem, Node* dest,
2590                                    Node* start_offset,
2591                                    Node* end_offset,
2592                                    PhaseGVN* phase) {
2593   if (start_offset == end_offset) {
2594     // nothing to do
2595     return mem;
2596   }
```



```
2598   Compile* C = phase->C;
2599   int unit = BytesPerLong;
2600   Node* zbase = start_offset;
2601   Node* zend  = end_offset;

2603   // Scale to the unit required by the CPU:
2604   if (!Matcher::init_array_count_is_in_bytes) {
2605     Node* shift = phase->intcon(exact_log2(unit));
2606     zbase = phase->transform( new(C,3) URShiftXNode(zbase, shift) );
2607     zend  = phase->transform( new(C,3) URShiftXNode(zend,  shift) );
2608   }

2610   Node* zsize = phase->transform( new(C,3) SubXNode(zend, zbase) );
2611   Node* zinit = phase->zerocon((unit == BytesPerLong) ? T_LONG : T_INT);

2613   // Bulk clear double-words
2614   Node* adr = phase->transform( new(C,4) AddPNode(dest, dest, start_offset) );
2615   mem = new (C, 4) ClearArrayNode(ctl, mem, zsize, adr);
2616   return phase->transform(mem);
2617 }

2619 Node* ClearArrayNode::clear_memory(Node* ctl, Node* mem, Node* dest,
2620                                    intptr_t start_offset,
2621                                    intptr_t end_offset,
2622                                    PhaseGVN* phase) {
2623   if (start_offset == end_offset) {
2624     // nothing to do
2625     return mem;
2626   }

2628   Compile* C = phase->C;
2629   assert((end_offset % BytesPerInt) == 0, "odd end offset");
2630   intptr_t done_offset = end_offset;
2631   if ((done_offset % BytesPerLong) != 0) {
2632     done_offset -= BytesPerInt;
2633   }
2634   if (done_offset > start_offset) {
2635     mem = clear_memory(ctl, mem, dest,
2636                        start_offset, phase->MakeConX(done_offset), phase);
2637   }
2638   if (done_offset < end_offset) { // emit the final 32-bit store
2639     Node* adr = new (C, 4) AddPNode(dest, dest, phase->MakeConX(done_offset));
2640     adr = phase->transform(adr);
2641     const TypePtr* atp = TypeRawPtr::BOTTOM;
2642     mem = StoreNode::make(*phase, ctl, mem, adr, atp, phase->zerocon(T_INT), T_I
2643     mem = phase->transform(mem);
2644     done_offset += BytesPerInt;
2645   }
2646   assert(done_offset == end_offset, "");
2647   return mem;
2648 }

2650 //=============================================================================
2651 // Do not match memory edge.
2652 uint StrIntrinsicNode::match_edge(uint idx) const {
2653   return idx == 2 || idx == 3;
2654 }

2656 //------------------------------Ideal------------------------------------------
2657 // Return a node which is more "ideal" than the current node.  Strip out
2658 // control copies
2659 Node *StrIntrinsicNode::Ideal(PhaseGVN *phase, bool can_reshape) {
2660   if (remove_dead_region(phase, can_reshape)) return this;

2662   if (can_reshape) {
```

```
2663     Node* mem = phase->transform(in(MemNode::Memory));
2664     // If transformed to a MergeMem, get the desired slice
2665     uint alias_idx = phase->C->get_alias_index(adr_type());
2666     mem = mem->is_MergeMem() ? mem->as_MergeMem()->memory_at(alias_idx) : mem;
2667     if (mem != in(MemNode::Memory)) {
2668       set_req(MemNode::Memory, mem);
2669       return this;
2670     }
2671   }
2672   return NULL;
2673 }

2675 //=============================================================================
2676 MemBarNode::MemBarNode(Compile* C, int alias_idx, Node* precedent)
2677   : MultiNode(TypeFunc::Parms + (precedent == NULL? 0: 1)),
2678     _adr_type(C->get_adr_type(alias_idx))
2679 {
2680   init_class_id(Class_MemBar);
2681   Node* top = C->top();
2682   init_req(TypeFunc::I_O,top);
2683   init_req(TypeFunc::FramePtr,top);
2684   init_req(TypeFunc::ReturnAdr,top);
2685   if (precedent != NULL)
2686     init_req(TypeFunc::Parms, precedent);
2687 }

2689 //------------------------------cmp--------------------------------------------
2690 uint MemBarNode::hash() const { return NO_HASH; }
2691 uint MemBarNode::cmp( const Node &n ) const {
2692   return (&n == this);          // Always fail except on self
2693 }

2695 //------------------------------make-------------------------------------------
2696 MemBarNode* MemBarNode::make(Compile* C, int opcode, int atp, Node* pn) {
2697   int len = Precedent + (pn == NULL? 0: 1);
2698   switch (opcode) {
2699   case Op_MemBarAcquire:   return new(C, len) MemBarAcquireNode(C,  atp, pn);
2700   case Op_MemBarRelease:   return new(C, len) MemBarReleaseNode(C,  atp, pn);
2701   case Op_MemBarAcquireLock: return new(C, len) MemBarAcquireLockNode(C,  atp, p
2702   case Op_MemBarReleaseLock: return new(C, len) MemBarReleaseLockNode(C,  atp, p
2703   case Op_MemBarVolatile:  return new(C, len) MemBarVolatileNode(C, atp, pn);
2704   case Op_MemBarCPUOrder:  return new(C, len) MemBarCPUOrderNode(C, atp, pn);
2705   case Op_Initialize:      return new(C, len) InitializeNode(C,     atp, pn);
2706   default:                 ShouldNotReachHere(); return NULL;
2707   }
2708 }

2710 //------------------------------Ideal------------------------------------------
2711 // Return a node which is more "ideal" than the current node.  Strip out
2712 // control copies
2713 Node *MemBarNode::Ideal(PhaseGVN *phase, bool can_reshape) {
2714   if (remove_dead_region(phase, can_reshape)) return this;

2716   // Eliminate volatile MemBars for scalar replaced objects.
2717   if (can_reshape && req() == (Precedent+1) &&
2718       (Opcode() == Op_MemBarAcquire || Opcode() == Op_MemBarVolatile)) {
2719     // Volatile field loads and stores.
2720     Node* my_mem = in(MemBarNode::Precedent);
2721     if (my_mem != NULL && my_mem->is_Mem()) {
2722       const TypeOopPtr* t_oop = my_mem->in(MemNode::Address)->bottom_type()->isa
2723       // Check for scalar replaced object reference.
2724       if( t_oop != NULL && t_oop->is_known_instance_field() &&
2725           t_oop->offset() != Type::OffsetBot &&
2726           t_oop->offset() != Type::OffsetTop) {
2727         // Replace MemBar projections by its inputs.
2728         PhaseIterGVN* igvn = phase->is_IterGVN();
```



```
2729         igvn->replace_node(proj_out(TypeFunc::Memory), in(TypeFunc::Memory));
2730         igvn->replace_node(proj_out(TypeFunc::Control), in(TypeFunc::Control));
2731         // Must return either the original node (now dead) or a new node
2732         // (Do not return a top here, since that would break the uniqueness of t
2733         return new (phase->C, 1) ConINode(TypeInt::ZERO);
2734       }
2735     }
2736   }
2737   return NULL;
2738 }

2740 //------------------------------Value------------------------------------------
2741 const Type *MemBarNode::Value( PhaseTransform *phase ) const {
2742   if( !in(0) ) return Type::TOP;
2743   if( phase->type(in(0)) == Type::TOP )
2744     return Type::TOP;
2745   return TypeTuple::MEMBAR;
2746 }

2748 //------------------------------match------------------------------------------
2749 // Construct projections for memory.
2750 Node *MemBarNode::match( const ProjNode *proj, const Matcher *m ) {
2751   switch (proj->_con) {
2752   case TypeFunc::Control:
2753   case TypeFunc::Memory:
2754     return new (m->C, 1) MachProjNode(this,proj->_con,RegMask::Empty,MachProjNod
2755   }
2756   ShouldNotReachHere();
2757   return NULL;
2758 }

2760 //==========================InitializeNode=====================================
2761 // SUMMARY:
2762 // This node acts as a memory barrier on raw memory, after some raw stores.
2763 // The 'cooked' oop value feeds from the Initialize, not the Allocation.
2764 // The Initialize can 'capture' suitably constrained stores as raw inits.
2765 // It can coalesce related raw stores into larger units (called 'tiles').
2766 // It can avoid zeroing new storage for memory units which have raw inits.
2767 // At macro-expansion, it is marked 'complete', and does not optimize further.
2768 //
2769 // EXAMPLE:
2770 // The object 'new short[2]' occupies 16 bytes in a 32-bit machine.
2771 //   ctl = incoming control; mem* = incoming memory
2772 // (Note:  A star * on a memory edge denotes I/O and other standard edges.)
2773 // First allocate uninitialized memory and fill in the header:
2774 //   alloc = (Allocate ctl mem* 16 #short[].klass ...)
2775 //   ctl := alloc.Control; mem* := alloc.Memory*
2776 //   rawmem = alloc.Memory; rawoop = alloc.RawAddress
2777 // Then initialize to zero the non-header parts of the raw memory block:
2778 //   init = (Initialize alloc.Control alloc.Memory* alloc.RawAddress)
2779 //   ctl := init.Control; mem.SLICE(#short[*]) := init.Memory
2780 // After the initialize node executes, the object is ready for service:
2781 //   oop := (CheckCastPP init.Control alloc.RawAddress #short[])
2782 // Suppose its body is immediately initialized as {1,2}:
2783 //   store1 = (StoreC init.Control init.Memory (+ oop 12) 1)
2784 //   store2 = (StoreC init.Control store1      (+ oop 14) 2)
2785 //   mem.SLICE(#short[*]) := store2
2786 //
2787 // DETAILS:
2788 // An InitializeNode collects and isolates object initialization after
2789 // an AllocateNode and before the next possible safepoint.  As a
2790 // memory barrier (MemBarNode), it keeps critical stores from drifting
2791 // down past any safepoint or any publication of the allocation.
2792 // Before this barrier, a newly-allocated object may have uninitialized bits.
2793 // After this barrier, it may be treated as a real oop, and GC is allowed.
2794 //
```

```
2795 // The semantics of the InitializeNode include an implicit zeroing of
2796 // the new object from object header to the end of the object.
2797 // (The object header and end are determined by the AllocateNode.)
2798 //
2799 // Certain stores may be added as direct inputs to the InitializeNode.
2800 // These stores must update raw memory, and they must be to addresses
2801 // derived from the raw address produced by AllocateNode, and with
2802 // a constant offset.  They must be ordered by increasing offset.
2803 // The first one is at in(RawStores), the last at in(req()-1).
2804 // Unlike most memory operations, they are not linked in a chain,
2805 // but are displayed in parallel as users of the rawmem output of
2806 // the allocation.
2807 //
2808 // (See comments in InitializeNode::capture_store, which continue
2809 // the example given above.)
2810 //
2811 // When the associated Allocate is macro-expanded, the InitializeNode
2812 // may be rewritten to optimize collected stores.  A ClearArrayNode
2813 // may also be created at that point to represent any required zeroing.
2814 // The InitializeNode is then marked 'complete', prohibiting further
2815 // capturing of nearby memory operations.
2816 //
2817 // During macro-expansion, all captured initializations which store
2818 // constant values of 32 bits or smaller are coalesced (if advantageous)
2819 // into larger 'tiles' 32 or 64 bits.  This allows an object to be
2820 // initialized in fewer memory operations.  Memory words which are
2821 // covered by neither tiles nor non-constant stores are pre-zeroed
2822 // by explicit stores of zero.  (The code shape happens to do all
2823 // zeroing first, then all other stores, with both sequences occurring
2824 // in order of ascending offsets.)
2825 //
2826 // Alternatively, code may be inserted between an AllocateNode and its
2827 // InitializeNode, to perform arbitrary initialization of the new object.
2828 // E.g., the object copying intrinsics insert complex data transfers here.
2829 // The initialization must then be marked as 'complete' disable the
2830 // built-in zeroing semantics and the collection of initializing stores.
2831 //
2832 // While an InitializeNode is incomplete, reads from the memory state
2833 // produced by it are optimizable if they match the control edge and
2834 // new oop address associated with the allocation/initialization.
2835 // They return a stored value (if the offset matches) or else zero.
2836 // A write to the memory state, if it matches control and address,
2837 // and if it is to a constant offset, may be 'captured' by the
2838 // InitializeNode.  It is cloned as a raw memory operation and rewired
2839 // inside the initialization, to the raw oop produced by the allocation.
2840 // Operations on addresses which are provably distinct (e.g., to
2841 // other AllocateNodes) are allowed to bypass the initialization.
2842 //
2843 // The effect of all this is to consolidate object initialization
2844 // (both arrays and non-arrays, both piecewise and bulk) into a
2845 // single location, where it can be optimized as a unit.
2846 //
2847 // Only stores with an offset less than TrackedInitializationLimit words
2848 // will be considered for capture by an InitializeNode.  This puts a
2849 // reasonable limit on the complexity of optimized initializations.

2851 //----------------------------InitializeNode------------------------------------
2852 InitializeNode::InitializeNode(Compile* C, int adr_type, Node* rawoop)
2853   : _is_complete(false),
2854     MemBarNode(C, adr_type, rawoop)
2855 {
2856   init_class_id(Class_Initialize);

2858   assert(adr_type == Compile::AliasIdxRaw, "only valid atp");
2859   assert(in(RawAddress) == rawoop, "proper init");
2860   // Note:  allocation() can be NULL, for secondary initialization barriers
```



```
2861 }

2863 // Since this node is not matched, it will be processed by the
2864 // register allocator.  Declare that there are no constraints
2865 // on the allocation of the RawAddress edge.
2866 const RegMask &InitializeNode::in_RegMask(uint idx) const {
2867   // This edge should be set to top, by the set_complete.  But be conservative.
2868   if (idx == InitializeNode::RawAddress)
2869     return *(Compile::current()->matcher()->idealreg2spillmask[in(idx)->ideal_re
2870   return RegMask::Empty;
2871 }

2873 Node* InitializeNode::memory(uint alias_idx) {
2874   Node* mem = in(Memory);
2875   if (mem->is_MergeMem()) {
2876     return mem->as_MergeMem()->memory_at(alias_idx);
2877   } else {
2878     // incoming raw memory is not split
2879     return mem;
2880   }
2881 }

2883 bool InitializeNode::is_non_zero() {
2884   if (is_complete())  return false;
2885   remove_extra_zeroes();
2886   return (req() > RawStores);
2887 }

2889 void InitializeNode::set_complete(PhaseGVN* phase) {
2890   assert(!is_complete(), "caller responsibility");
2891   _is_complete = true;

2893   // After this node is complete, it contains a bunch of
2894   // raw-memory initializations.  There is no need for
2895   // it to have anything to do with non-raw memory effects.
2896   // Therefore, tell all non-raw users to re-optimize themselves,
2897   // after skipping the memory effects of this initialization.
2898   PhaseIterGVN* igvn = phase->is_IterGVN();
2899   if (igvn)  igvn->add_users_to_worklist(this);
2900 }

2902 // convenience function
2903 // return false if the init contains any stores already
2904 bool AllocateNode::maybe_set_complete(PhaseGVN* phase) {
2905   InitializeNode* init = initialization();
2906   if (init == NULL || init->is_complete())  return false;
2907   init->remove_extra_zeroes();
2908   // for now, if this allocation has already collected any inits, bail:
2909   if (init->is_non_zero())  return false;
2910   init->set_complete(phase);
2911   return true;
2912 }

2914 void InitializeNode::remove_extra_zeroes() {
2915   if (req() == RawStores)  return;
2916   Node* zmem = zero_memory();
2917   uint fill = RawStores;
2918   for (uint i = fill; i < req(); i++) {
2919     Node* n = in(i);
2920     if (n->is_top() || n == zmem)  continue;  // skip
2921     if (fill < i)  set_req(fill, n);          // compact
2922     ++fill;
2923   }
2924   // delete any empty spaces created:
2925   while (fill < req()) {
2926     del_req(fill);
```

```
2927   }
2928 }

2930 // Helper for remembering which stores go with which offsets.
2931 intptr_t InitializeNode::get_store_offset(Node* st, PhaseTransform* phase) {
2932   if (!st->is_Store())  return -1;  // can happen to dead code via subsume_node
2933   intptr_t offset = -1;
2934   Node* base = AddPNode::Ideal_base_and_offset(st->in(MemNode::Address),
2935                                                phase, offset);
2936   if (base == NULL)     return -1;  // something is dead,
2937   if (offset < 0)       return -1;  //        dead, dead
2938   return offset;
2939 }

2941 // Helper for proving that an initialization expression is
2942 // "simple enough" to be folded into an object initialization.
2943 // Attempts to prove that a store's initial value 'n' can be captured
2944 // within the initialization without creating a vicious cycle, such as:
2945 //     { Foo p = new Foo(); p.next = p; }
2946 // True for constants and parameters and small combinations thereof.
2947 bool InitializeNode::detect_init_independence(Node* n,
2948                                               bool st_is_pinned,
2949                                               int& count) {
2950   if (n == NULL)      return true;   // (can this really happen?)
2951   if (n->is_Proj())   n = n->in(0);
2952   if (n == this)      return false;  // found a cycle
2953   if (n->is_Con())    return true;
2954   if (n->is_Start())  return true;   // params, etc., are OK
2955   if (n->is_Root())   return true;   // even better

2957   Node* ctl = n->in(0);
2958   if (ctl != NULL && !ctl->is_top()) {
2959     if (ctl->is_Proj())  ctl = ctl->in(0);
2960     if (ctl == this)  return false;

2962     // If we already know that the enclosing memory op is pinned right after
2963     // the init, then any control flow that the store has picked up
2964     // must have preceded the init, or else be equal to the init.
2965     // Even after loop optimizations (which might change control edges)
2966     // a store is never pinned *before* the availability of its inputs.
2967     if (!MemNode::all_controls_dominate(n, this))
2968       return false;                  // failed to prove a good control

2970   }

2972   // Check data edges for possible dependencies on 'this'.
2973   if ((count += 1) > 20)  return false;  // complexity limit
2974   for (uint i = 1; i < n->req(); i++) {
2975     Node* m = n->in(i);
2976     if (m == NULL || m == n || m->is_top())  continue;
2977     uint first_i = n->find_edge(m);
2978     if (i != first_i)  continue;  // process duplicate edge just once
2979     if (!detect_init_independence(m, st_is_pinned, count)) {
2980       return false;
2981     }
2982   }

2984   return true;
2985 }

2987 // Here are all the checks a Store must pass before it can be moved into
2988 // an initialization.  Returns zero if a check fails.
2989 // On success, returns the (constant) offset to which the store applies,
2990 // within the initialized memory.
2991 intptr_t InitializeNode::can_capture_store(StoreNode* st, PhaseTransform* phase)
2992   const int FAIL = 0;
```



```
2993   if (st->req() != MemNode::ValueIn + 1)
2994     return FAIL;                // an inscrutable StoreNode (card mark?)
2995   Node* ctl = st->in(MemNode::Control);
2996   if (!(ctl != NULL && ctl->is_Proj() && ctl->in(0) == this))
2997     return FAIL;                // must be unconditional after the initializatio
2998   Node* mem = st->in(MemNode::Memory);
2999   if (!(mem->is_Proj() && mem->in(0) == this))
3000     return FAIL;                // must not be preceded by other stores
3001   Node* adr = st->in(MemNode::Address);
3002   intptr_t offset;
3003   AllocateNode* alloc = AllocateNode::Ideal_allocation(adr, phase, offset);
3004   if (alloc == NULL)
3005     return FAIL;                // inscrutable address
3006   if (alloc != allocation())
3007     return FAIL;                // wrong allocation!  (store needs to float up)
3008   Node* val = st->in(MemNode::ValueIn);
3009   int complexity_count = 0;
3010   if (!detect_init_independence(val, true, complexity_count))
3011     return FAIL;                // stored value must be 'simple enough'

3013   return offset;                // success
3014 }

3016 // Find the captured store in(i) which corresponds to the range
3017 // [start..start+size) in the initialized object.
3018 // If there is one, return its index i.  If there isn't, return the
3019 // negative of the index where it should be inserted.
3020 // Return 0 if the queried range overlaps an initialization boundary
3021 // or if dead code is encountered.
3022 // If size_in_bytes is zero, do not bother with overlap checks.
3023 int InitializeNode::captured_store_insertion_point(intptr_t start,
3024                                                    int size_in_bytes,
3025                                                    PhaseTransform* phase) {
3026   const int FAIL = 0, MAX_STORE = BytesPerLong;

3028   if (is_complete())
3029     return FAIL;                // arraycopy got here first; punt

3031   assert(allocation() != NULL, "must be present");

3033   // no negatives, no header fields:
3034   if (start < (intptr_t) allocation()->minimum_header_size())  return FAIL;

3036   // after a certain size, we bail out on tracking all the stores:
3037   intptr_t ti_limit = (TrackedInitializationLimit * HeapWordSize);
3038   if (start >= ti_limit)  return FAIL;

3040   for (uint i = InitializeNode::RawStores, limit = req(); ; ) {
3041     if (i >= limit)  return -(int)i; // not found; here is where to put it

3043     Node*    st     = in(i);
3044     intptr_t st_off = get_store_offset(st, phase);
3045     if (st_off < 0) {
3046       if (st != zero_memory()) {
3047         return FAIL;            // bail out if there is dead garbage
3048       }
3049     } else if (st_off > start) {
3050       // ...we are done, since stores are ordered
3051       if (st_off < start + size_in_bytes) {
3052         return FAIL;            // the next store overlaps
3053       }
3054       return -(int)i;           // not found; here is where to put it
3055     } else if (st_off < start) {
3056       if (size_in_bytes != 0 &&
3057           start < st_off + MAX_STORE &&
3058           start < st_off + st->as_Store()->memory_size()) {
```

```
3059         return FAIL;            // the previous store overlaps
3060       }
3061     } else {
3062       if (size_in_bytes != 0 &&
3063           st->as_Store()->memory_size() != size_in_bytes) {
3064         return FAIL;            // mismatched store size
3065       }
3066       return i;
3067     }
3068
3069     ++i;
3070   }
3071 }
3072
3073 // Look for a captured store which initializes at the offset 'start'
3074 // with the given size.  If there is no such store, and no other
3075 // initialization interferes, then return zero_memory (the memory
3076 // projection of the AllocateNode).
3077 Node* InitializeNode::find_captured_store(intptr_t start, int size_in_bytes,
3078                                           PhaseTransform* phase) {
3079   assert(stores_are_sane(phase), "");
3080   int i = captured_store_insertion_point(start, size_in_bytes, phase);
3081   if (i == 0) {
3082     return NULL;                // something is dead
3083   } else if (i < 0) {
3084     return zero_memory();       // just primordial zero bits here
3085   } else {
3086     Node* st = in(i);           // here is the store at this position
3087     assert(get_store_offset(st->as_Store(), phase) == start, "sanity");
3088     return st;
3089   }
3090 }
3091
3092 // Create, as a raw pointer, an address within my new object at 'offset'.
3093 Node* InitializeNode::make_raw_address(intptr_t offset,
3094                                        PhaseTransform* phase) {
3095   Node* addr = in(RawAddress);
3096   if (offset != 0) {
3097     Compile* C = phase->C;
3098     addr = phase->transform( new (C, 4) AddPNode(C->top(), addr,
3099                                                  phase->MakeConX(offset)) );
3100   }
3101   return addr;
3102 }
3103
3104 // Clone the given store, converting it into a raw store
3105 // initializing a field or element of my new object.
3106 // Caller is responsible for retiring the original store,
3107 // with subsume_node or the like.
3108 //
3109 // From the example above InitializeNode::InitializeNode,
3110 // here are the old stores to be captured:
3111 //   store1 = (StoreC init.Control init.Memory (+ oop 12) 1)
3112 //   store2 = (StoreC init.Control store1      (+ oop 14) 2)
3113 //
3114 // Here is the changed code; note the extra edges on init:
3115 //   alloc = (Allocate ...)
3116 //   rawoop = alloc.RawAddress
3117 //   rawstore1 = (StoreC alloc.Control alloc.Memory (+ rawoop 12) 1)
3118 //   rawstore2 = (StoreC alloc.Control alloc.Memory (+ rawoop 14) 2)
3119 //   init = (Initialize alloc.Control alloc.Memory rawoop
3120 //                      rawstore1 rawstore2)
3121 //
3122 Node* InitializeNode::capture_store(StoreNode* st, intptr_t start,
3123                                     PhaseTransform* phase) {
3124   assert(stores_are_sane(phase), "");
```



```
3125
3126   if (start < 0)  return NULL;
3127   assert(can_capture_store(st, phase) == start, "sanity");
3128
3129   Compile* C = phase->C;
3130   int size_in_bytes = st->memory_size();
3131   int i = captured_store_insertion_point(start, size_in_bytes, phase);
3132   if (i == 0)  return NULL;     // bail out
3133   Node* prev_mem = NULL;        // raw memory for the captured store
3134   if (i > 0) {
3135     prev_mem = in(i);           // there is a pre-existing store under this one
3136     set_req(i, C->top());       // temporarily disconnect it
3137     // See StoreNode::Ideal 'st->outcnt() == 1' for the reason to disconnect.
3138   } else {
3139     i = -i;                     // no pre-existing store
3140     prev_mem = zero_memory();   // a slice of the newly allocated object
3141     if (i > InitializeNode::RawStores && in(i-1) == prev_mem)
3142       set_req(--i, C->top());   // reuse this edge; it has been folded away
3143     else
3144       ins_req(i, C->top());     // build a new edge
3145   }
3146   Node* new_st = st->clone();
3147   new_st->set_req(MemNode::Control, in(Control));
3148   new_st->set_req(MemNode::Memory,  prev_mem);
3149   new_st->set_req(MemNode::Address, make_raw_address(start, phase));
3150   new_st = phase->transform(new_st);
3151
3152   // At this point, new_st might have swallowed a pre-existing store
3153   // at the same offset, or perhaps new_st might have disappeared,
3154   // if it redundantly stored the same value (or zero to fresh memory).
3155
3156   // In any case, wire it in:
3157   set_req(i, new_st);
3158
3159   // The caller may now kill the old guy.
3160   DEBUG_ONLY(Node* check_st = find_captured_store(start, size_in_bytes, phase));
3161   assert(check_st == new_st || check_st == NULL, "must be findable");
3162   assert(!is_complete(), "");
3163   return new_st;
3164 }
3165
3166 static bool store_constant(jlong* tiles, int num_tiles,
3167                            intptr_t st_off, int st_size,
3168                            jlong con) {
3169   if ((st_off & (st_size-1)) != 0)
3170     return false;               // strange store offset (assume size==2**N)
3171   address addr = (address)tiles + st_off;
3172   assert(st_off >= 0 && addr+st_size <= (address)&tiles[num_tiles], "oob");
3173   switch (st_size) {
3174   case sizeof(jbyte):  *(jbyte*) addr = (jbyte) con; break;
3175   case sizeof(jchar):  *(jchar*) addr = (jchar) con; break;
3176   case sizeof(jint):   *(jint*)  addr = (jint)  con; break;
3177   case sizeof(jlong):  *(jlong*) addr = (jlong) con; break;
3178   default: return false;        // strange store size (detect size!=2**N here)
3179   }
3180   return true;                  // return success to caller
3181 }
3182
3183 // Coalesce subword constants into int constants and possibly
3184 // into long constants.  The goal, if the CPU permits,
3185 // is to initialize the object with a small number of 64-bit tiles.
3186 // Also, convert floating-point constants to bit patterns.
3187 // Non-constants are not relevant to this pass.
3188 //
3189 // In terms of the running example on InitializeNode::InitializeNode
3190 // and InitializeNode::capture_store, here is the transformation
```

```
3191 // of rawstore1 and rawstore2 into rawstore12:
3192 //   alloc = (Allocate ...)
3193 //   rawoop = alloc.RawAddress
3194 //   tile12 = 0x00010002
3195 //   rawstore12 = (StoreI alloc.Control alloc.Memory (+ rawoop 12) tile12)
3196 //   init = (Initialize alloc.Control alloc.Memory rawoop rawstore12)
3197 //
3198 void
3199 InitializeNode::coalesce_subword_stores(intptr_t header_size,
3200                                         Node* size_in_bytes,
3201                                         PhaseGVN* phase) {
3202   Compile* C = phase->C;

3204   assert(stores_are_sane(phase), "");
3205   // Note:  After this pass, they are not completely sane,
3206   // since there may be some overlaps.

3208   int old_subword = 0, old_long = 0, new_int = 0, new_long = 0;

3210   intptr_t ti_limit = (TrackedInitializationLimit * HeapWordSize);
3211   intptr_t size_limit = phase->find_intptr_t_con(size_in_bytes, ti_limit);
3212   size_limit = MIN2(size_limit, ti_limit);
3213   size_limit = align_size_up(size_limit, BytesPerLong);
3214   int num_tiles = size_limit / BytesPerLong;

3216   // allocate space for the tile map:
3217   const int small_len = DEBUG_ONLY(true ? 3 :) 30; // keep stack frames small
3218   jlong  tiles_buf[small_len];
3219   Node*  nodes_buf[small_len];
3220   jlong  inits_buf[small_len];
3221   jlong* tiles = ((num_tiles <= small_len) ? &tiles_buf[0]
3222                   : NEW_RESOURCE_ARRAY(jlong, num_tiles));
3223   Node** nodes = ((num_tiles <= small_len) ? &nodes_buf[0]
3224                   : NEW_RESOURCE_ARRAY(Node*, num_tiles));
3225   jlong* inits = ((num_tiles <= small_len) ? &inits_buf[0]
3226                   : NEW_RESOURCE_ARRAY(jlong, num_tiles));
3227   // tiles: exact bitwise model of all primitive constants
3228   // nodes: last constant-storing node subsumed into the tiles model
3229   // inits: which bytes (in each tile) are touched by any initializations

3231   //// Pass A: Fill in the tile model with any relevant stores.

3233   Copy::zero_to_bytes(tiles, sizeof(tiles[0]) * num_tiles);
3234   Copy::zero_to_bytes(nodes, sizeof(nodes[0]) * num_tiles);
3235   Copy::zero_to_bytes(inits, sizeof(inits[0]) * num_tiles);
3236   Node* zmem = zero_memory(); // initially zero memory state
3237   for (uint i = InitializeNode::RawStores, limit = req(); i < limit; i++) {
3238     Node* st = in(i);
3239     intptr_t st_off = get_store_offset(st, phase);

3241     // Figure out the store's offset and constant value:
3242     if (st_off < header_size)             continue; //skip (ignore header)
3243     if (st->in(MemNode::Memory) != zmem)  continue; //skip (odd store chain)
3244     int st_size = st->as_Store()->memory_size();
3245     if (st_off + st_size > size_limit)    break;

3247     // Record which bytes are touched, whether by constant or not.
3248     if (!store_constant(inits, num_tiles, st_off, st_size, (jlong) -1))
3249       continue;                 // skip (strange store size)

3251     const Type* val = phase->type(st->in(MemNode::ValueIn));
3252     if (!val->singleton())                continue; //skip (non-con store)
3253     BasicType type = val->basic_type();

3255     jlong con = 0;
3256     switch (type) {
```



```
3257     case T_INT:    con = val->is_int()->get_con();  break;
3258     case T_LONG:   con = val->is_long()->get_con(); break;
3259     case T_FLOAT:  con = jint_cast(val->getf());    break;
3260     case T_DOUBLE: con = jlong_cast(val->getd());   break;
3261     default:                              continue; //skip (odd store type)
3262     }

3264     if (type == T_LONG && Matcher::isSimpleConstant64(con) &&
3265         st->Opcode() == Op_StoreL) {
3266       continue;                 // This StoreL is already optimal.
3267     }

3269     // Store down the constant.
3270     store_constant(tiles, num_tiles, st_off, st_size, con);

3272     intptr_t j = st_off >> LogBytesPerLong;

3274     if (type == T_INT && st_size == BytesPerInt
3275         && (st_off & BytesPerInt) == BytesPerInt) {
3276       jlong lcon = tiles[j];
3277       if (!Matcher::isSimpleConstant64(lcon) &&
3278           st->Opcode() == Op_StoreI) {
3279         // This StoreI is already optimal by itself.
3280         jint* intcon = (jint*) &tiles[j];
3281         intcon[1] = 0;  // undo the store_constant()

3283         // If the previous store is also optimal by itself, back up and
3284         // undo the action of the previous loop iteration... if we can.
3285         // But if we can't, just let the previous half take care of itself.
3286         st = nodes[j];
3287         st_off -= BytesPerInt;
3288         con = intcon[0];
3289         if (con != 0 && st != NULL && st->Opcode() == Op_StoreI) {
3290           assert(st_off >= header_size, "still ignoring header");
3291           assert(get_store_offset(st, phase) == st_off, "must be");
3292           assert(in(i-1) == zmem, "must be");
3293           DEBUG_ONLY(const Type* tcon = phase->type(st->in(MemNode::ValueIn)));
3294           assert(con == tcon->is_int()->get_con(), "must be");
3295           // Undo the effects of the previous loop trip, which swallowed st:
3296           intcon[0] = 0;        // undo store_constant()
3297           set_req(i-1, st);     // undo set_req(i, zmem)
3298           nodes[j] = NULL;      // undo nodes[j] = st
3299           --old_subword;        // undo ++old_subword
3300         }
3301         continue;               // This StoreI is already optimal.
3302       }
3303     }

3305     // This store is not needed.
3306     set_req(i, zmem);
3307     nodes[j] = st;              // record for the moment
3308     if (st_size < BytesPerLong) // something has changed
3309         ++old_subword;          // includes int/float, but who's counting...
3310     else  ++old_long;
3311   }

3313   if ((old_subword + old_long) == 0)
3314     return;                     // nothing more to do

3316   //// Pass B: Convert any non-zero tiles into optimal constant stores.
3317   // Be sure to insert them before overlapping non-constant stores.
3318   // (E.g., byte[] x = { 1,2,y,4 }  =>  x[int 0] = 0x01020004, x[2]=y.)
3319   for (int j = 0; j < num_tiles; j++) {
3320     jlong con  = tiles[j];
3321     jlong init = inits[j];
3322     if (con == 0)  continue;
```

```
3323     jint con0,  con1;           // split the constant, address-wise
3324     jint init0, init1;          // split the init map, address-wise
3325     { union { jlong con; jint intcon[2]; } u;
3326       u.con = con;
3327       con0  = u.intcon[0];
3328       con1  = u.intcon[1];
3329       u.con = init;
3330       init0 = u.intcon[0];
3331       init1 = u.intcon[1];
3332     }

3334     Node* old = nodes[j];
3335     assert(old != NULL, "need the prior store");
3336     intptr_t offset = (j * BytesPerLong);

3338     bool split = !Matcher::isSimpleConstant64(con);

3340     if (offset < header_size) {
3341       assert(offset + BytesPerInt >= header_size, "second int counts");
3342       assert(*(jint*)&tiles[j] == 0, "junk in header");
3343       split = true;             // only the second word counts
3344       // Example:  int a[] = { 42 ... }
3345     } else if (con0 == 0 && init0 == -1) {
3346       split = true;             // first word is covered by full inits
3347       // Example:  int a[] = { ... foo(), 42 ... }
3348     } else if (con1 == 0 && init1 == -1) {
3349       split = true;             // second word is covered by full inits
3350       // Example:  int a[] = { ... 42, foo() ... }
3351     }

3353     // Here's a case where init0 is neither 0 nor -1:
3354     //   byte a[] = { ... 0,0,foo(),0,  0,0,0,42 ... }
3355     // Assuming big-endian memory, init0, init1 are 0x0000FF00, 0x000000FF.
3356     // In this case the tile is not split; it is (jlong)42.
3357     // The big tile is stored down, and then the foo() value is inserted.
3358     // (If there were foo(),foo() instead of foo(),0, init0 would be -1.)

3360     Node* ctl = old->in(MemNode::Control);
3361     Node* adr = make_raw_address(offset, phase);
3362     const TypePtr* atp = TypeRawPtr::BOTTOM;

3364     // One or two coalesced stores to plop down.
3365     Node*    st[2];
3366     intptr_t off[2];
3367     int  nst = 0;
3368     if (!split) {
3369       ++new_long;
3370       off[nst] = offset;
3371       st[nst++] = StoreNode::make(*phase, ctl, zmem, adr, atp,
3372                                   phase->longcon(con), T_LONG);
3373     } else {
3374       // Omit either if it is a zero.
3375       if (con0 != 0) {
3376         ++new_int;
3377         off[nst]  = offset;
3378         st[nst++] = StoreNode::make(*phase, ctl, zmem, adr, atp,
3379                                     phase->intcon(con0), T_INT);
3380       }
3381       if (con1 != 0) {
3382         ++new_int;
3383         offset += BytesPerInt;
3384         adr = make_raw_address(offset, phase);
3385         off[nst]  = offset;
3386         st[nst++] = StoreNode::make(*phase, ctl, zmem, adr, atp,
3387                                     phase->intcon(con1), T_INT);
3388       }
```



```
3389     }

3391     // Insert second store first, then the first before the second.
3392     // Insert each one just before any overlapping non-constant stores.
3393     while (nst > 0) {
3394       Node* st1 = st[--nst];
3395       C->copy_node_notes_to(st1, old);
3396       st1 = phase->transform(st1);
3397       offset = off[nst];
3398       assert(offset >= header_size, "do not smash header");
3399       int ins_idx = captured_store_insertion_point(offset, /*size:*/0, phase);
3400       guarantee(ins_idx != 0, "must re-insert constant store");
3401       if (ins_idx < 0)  ins_idx = -ins_idx;  // never overlap
3402       if (ins_idx > InitializeNode::RawStores && in(ins_idx-1) == zmem)
3403         set_req(--ins_idx, st1);
3404       else
3405         ins_req(ins_idx, st1);
3406     }
3407   }

3409   if (PrintCompilation && WizardMode)
3410     tty->print_cr("Changed %d/%d subword/long constants into %d/%d int/long",
3411                   old_subword, old_long, new_int, new_long);
3412   if (C->log() != NULL)
3413     C->log()->elem("comment that='%d/%d subword/long to %d/%d int/long'",
3414                    old_subword, old_long, new_int, new_long);

3416   // Clean up any remaining occurrences of zmem:
3417   remove_extra_zeroes();
3418 }

3420 // Explore forward from in(start) to find the first fully initialized
3421 // word, and return its offset.  Skip groups of subword stores which
3422 // together initialize full words.  If in(start) is itself part of a
3423 // fully initialized word, return the offset of in(start).  If there
3424 // are no following full-word stores, or if something is fishy, return
3425 // a negative value.
3426 intptr_t InitializeNode::find_next_fullword_store(uint start, PhaseGVN* phase) {
3427   int       int_map = 0;
3428   intptr_t  int_map_off = 0;
3429   const int FULL_MAP = right_n_bits(BytesPerInt);  // the int_map we hope for

3431   for (uint i = start, limit = req(); i < limit; i++) {
3432     Node* st = in(i);

3434     intptr_t st_off = get_store_offset(st, phase);
3435     if (st_off < 0)  break;  // return conservative answer

3437     int st_size = st->as_Store()->memory_size();
3438     if (st_size >= BytesPerInt && (st_off % BytesPerInt) == 0) {
3439       return st_off;            // we found a complete word init
3440     }

3442     // update the map:

3444     intptr_t this_int_off = align_size_down(st_off, BytesPerInt);
3445     if (this_int_off != int_map_off) {
3446       // reset the map:
3447       int_map = 0;
3448       int_map_off = this_int_off;
3449     }

3451     int subword_off = st_off - this_int_off;
3452     int_map |= right_n_bits(st_size) << subword_off;
3453     if ((int_map & FULL_MAP) == FULL_MAP) {
3454       return this_int_off;      // we found a complete word init
```

```
3455     }
3457     // Did this store hit or cross the word boundary?
3458     intptr_t next_int_off = align_size_down(st_off + st_size, BytesPerInt);
3459     if (next_int_off == this_int_off + BytesPerInt) {
3460       // We passed the current int, without fully initializing it.
3461       int_map_off = next_int_off;
3462       int_map >>= BytesPerInt;
3463     } else if (next_int_off > this_int_off + BytesPerInt) {
3464       // We passed the current and next int.
3465       return this_int_off + BytesPerInt;
3466     }
3467   }
3469   return -1;
3470 }
3473 // Called when the associated AllocateNode is expanded into CFG.
3474 // At this point, we may perform additional optimizations.
3475 // Linearize the stores by ascending offset, to make memory
3476 // activity as coherent as possible.
3477 Node* InitializeNode::complete_stores(Node* rawctl, Node* rawmem, Node* rawptr,
3478                                       intptr_t header_size,
3479                                       Node* size_in_bytes,
3480                                       PhaseGVN* phase) {
3481   assert(!is_complete(), "not already complete");
3482   assert(stores_are_sane(phase), "");
3483   assert(allocation() != NULL, "must be present");
3485   remove_extra_zeroes();
3487   if (ReduceFieldZeroing || ReduceBulkZeroing)
3488     // reduce instruction count for common initialization patterns
3489     coalesce_subword_stores(header_size, size_in_bytes, phase);
3491   Node* zmem = zero_memory();   // initially zero memory state
3492   Node* inits = zmem;           // accumulating a linearized chain of inits
3493   #ifdef ASSERT
3494   intptr_t first_offset = allocation()->minimum_header_size();
3495   intptr_t last_init_off = first_offset;  // previous init offset
3496   intptr_t last_init_end = first_offset;  // previous init offset+size
3497   intptr_t last_tile_end = first_offset;  // previous tile offset+size
3498   #endif
3499   intptr_t zeroes_done = header_size;
3501   bool do_zeroing = true;       // we might give up if inits are very sparse
3502   int  big_init_gaps = 0;       // how many large gaps have we seen?
3504   if (ZeroTLAB)  do_zeroing = false;
3505   if (!ReduceFieldZeroing && !ReduceBulkZeroing)  do_zeroing = false;
3507   for (uint i = InitializeNode::RawStores, limit = req(); i < limit; i++) {
3508     Node* st = in(i);
3509     intptr_t st_off = get_store_offset(st, phase);
3510     if (st_off < 0)
3511       break;                    // unknown junk in the inits
3512     if (st->in(MemNode::Memory) != zmem)
3513       break;                    // complicated store chains somehow in list
3515     int st_size = st->as_Store()->memory_size();
3516     intptr_t next_init_off = st_off + st_size;
3518     if (do_zeroing && zeroes_done < next_init_off) {
3519       // See if this store needs a zero before it or under it.
3520       intptr_t zeroes_needed = st_off;
```



```
3522       if (st_size < BytesPerInt) {
3523         // Look for subword stores which only partially initialize words.
3524         // If we find some, we must lay down some word-level zeroes first,
3525         // underneath the subword stores.
3526         //
3527         // Examples:
3528         //   byte[] a = { p,q,r,s }  =>  a[0]=p,a[1]=q,a[2]=r,a[3]=s
3529         //   byte[] a = { x,y,0,0 }  =>  a[0..3] = 0, a[0]=x,a[1]=y
3530         //   byte[] a = { 0,0,z,0 }  =>  a[0..3] = 0, a[2]=z
3531         //
3532         // Note:  coalesce_subword_stores may have already done this,
3533         // if it was prompted by constant non-zero subword initializers.
3534         // But this case can still arise with non-constant stores.
3536         intptr_t next_full_store = find_next_fullword_store(i, phase);
3538         // In the examples above:
3539         //   in(i)          p   q   r   s     x   y     z
3540         //   st_off        12  13  14  15    12  13    14
3541         //   st_size        1   1   1   1     1   1     1
3542         //   next_full_s.  12  16  16  16    16  16    16
3543         //   z's_done      12  16  16  16    12  16    12
3544         //   z's_needed    12  16  16  16    16  16    16
3545         //   zsize          0   0   0   0     4   0     4
3546         if (next_full_store < 0) {
3547           // Conservative tack:  Zero to end of current word.
3548           zeroes_needed = align_size_up(zeroes_needed, BytesPerInt);
3549         } else {
3550           // Zero to beginning of next fully initialized word.
3551           // Or, don't zero at all, if we are already in that word.
3552           assert(next_full_store >= zeroes_needed, "must go forward");
3553           assert((next_full_store & (BytesPerInt-1)) == 0, "even boundary");
3554           zeroes_needed = next_full_store;
3555         }
3556       }
3558       if (zeroes_needed > zeroes_done) {
3559         intptr_t zsize = zeroes_needed - zeroes_done;
3560         // Do some incremental zeroing on rawmem, in parallel with inits.
3561         zeroes_done = align_size_down(zeroes_done, BytesPerInt);
3562         rawmem = ClearArrayNode::clear_memory(rawctl, rawmem, rawptr,
3563                                               zeroes_done, zeroes_needed,
3564                                               phase);
3565         zeroes_done = zeroes_needed;
3566         if (zsize > Matcher::init_array_short_size && ++big_init_gaps > 2)
3567           do_zeroing = false;   // leave the hole, next time
3568       }
3569     }
3571     // Collect the store and move on:
3572     st->set_req(MemNode::Memory, inits);
3573     inits = st;                 // put it on the linearized chain
3574     set_req(i, zmem);           // unhook from previous position
3576     if (zeroes_done == st_off)
3577       zeroes_done = next_init_off;
3579     assert(!do_zeroing || zeroes_done >= next_init_off, "don't miss any");
3581     #ifdef ASSERT
3582     // Various order invariants.  Weaker than stores_are_sane because
3583     // a large constant tile can be filled in by smaller non-constant stores.
3584     assert(st_off >= last_init_off, "inits do not reverse");
3585     last_init_off = st_off;
3586     const Type* val = NULL;
```

```
3587     if (st_size >= BytesPerInt &&
3588         (val = phase->type(st->in(MemNode::ValueIn)))->singleton() &&
3589         (int)val->basic_type() < (int)T_OBJECT) {
3590       assert(st_off >= last_tile_end, "tiles do not overlap");
3591       assert(st_off >= last_init_end, "tiles do not overwrite inits");
3592       last_tile_end = MAX2(last_tile_end, next_init_off);
3593     } else {
3594       intptr_t st_tile_end = align_size_up(next_init_off, BytesPerLong);
3595       assert(st_tile_end >= last_tile_end, "inits stay with tiles");
3596       assert(st_off      >= last_init_end, "inits do not overlap");
3597       last_init_end = next_init_off;  // it's a non-tile
3598     }
3599     #endif //ASSERT
3600   }

3602   remove_extra_zeroes();        // clear out all the zmems left over
3603   add_req(inits);

3605   if (!ZeroTLAB) {
3606     // If anything remains to be zeroed, zero it all now.
3607     zeroes_done = align_size_down(zeroes_done, BytesPerInt);
3608     // if it is the last unused 4 bytes of an instance, forget about it
3609     intptr_t size_limit = phase->find_intptr_t_con(size_in_bytes, max_jint);
3610     if (zeroes_done + BytesPerLong >= size_limit) {
3611       assert(allocation() != NULL, "");
3612       if (allocation()->Opcode() == Op_Allocate) {
3613         Node* klass_node = allocation()->in(AllocateNode::KlassNode);
3614         ciKlass* k = phase->type(klass_node)->is_klassptr()->klass();
3615         if (zeroes_done == k->layout_helper())
3616           zeroes_done = size_limit;
3617       }
3618     }
3619     if (zeroes_done < size_limit) {
3620       rawmem = ClearArrayNode::clear_memory(rawctl, rawmem, rawptr,
3621                                             zeroes_done, size_in_bytes, phase);
3622     }
3623   }

3625   set_complete(phase);
3626   return rawmem;
3627 }


3630 #ifdef ASSERT
3631 bool InitializeNode::stores_are_sane(PhaseTransform* phase) {
3632   if (is_complete())
3633     return true;                // stores could be anything at this point
3634   assert(allocation() != NULL, "must be present");
3635   intptr_t last_off = allocation()->minimum_header_size();
3636   for (uint i = InitializeNode::RawStores; i < req(); i++) {
3637     Node* st = in(i);
3638     intptr_t st_off = get_store_offset(st, phase);
3639     if (st_off < 0)  continue;  // ignore dead garbage
3640     if (last_off > st_off) {
3641       tty->print_cr("*** bad store offset at %d: %d > %d", i, last_off, st_off);
3642       this->dump(2);
3643       assert(false, "ascending store offsets");
3644       return false;
3645     }
3646     last_off = st_off + st->as_Store()->memory_size();
3647   }
3648   return true;
3649 }
3650 #endif //ASSERT
```



```
3655 //============================MergeMemNode=====================================
3656 //
3657 // SEMANTICS OF MEMORY MERGES:  A MergeMem is a memory state assembled from seve
3658 // contributing store or call operations.  Each contributor provides the memory
3659 // state for a particular "alias type" (see Compile::alias_type).  For example,
3660 // if a MergeMem has an input X for alias category #6, then any memory reference
3661 // to alias category #6 may use X as its memory state input, as an exact equival
3662 // to using the MergeMem as a whole.
3663 //   Load<6>( MergeMem(<6>: X, ...), p ) <==> Load<6>(X,p)
3664 //
3665 // (Here, the <N> notation gives the index of the relevant adr_type.)
3666 //
3667 // In one special case (and more cases in the future), alias categories overlap.
3668 // The special alias category "Bot" (Compile::AliasIdxBot) includes all memory
3669 // states.  Therefore, if a MergeMem has only one contributing input W for Bot,
3670 // it is exactly equivalent to that state W:
3671 //   MergeMem(<Bot>: W) <==> W
3672 //
3673 // Usually, the merge has more than one input.  In that case, where inputs
3674 // overlap (i.e., one is Bot), the narrower alias type determines the memory
3675 // state for that type, and the wider alias type (Bot) fills in everywhere else:
3676 //   Load<5>( MergeMem(<Bot>: W, <6>: X), p ) <==> Load<5>(W,p)
3677 //   Load<6>( MergeMem(<Bot>: W, <6>: X), p ) <==> Load<6>(X,p)
3678 //
3679 // A merge can take a "wide" memory state as one of its narrow inputs.
3680 // This simply means that the merge observes out only the relevant parts of
3681 // the wide input.  That is, wide memory states arriving at narrow merge inputs
3682 // are implicitly "filtered" or "sliced" as necessary.  (This is rare.)
3683 //
3684 // These rules imply that MergeMem nodes may cascade (via their <Bot> links),
3685 // and that memory slices "leak through":
3686 //   MergeMem(<Bot>: MergeMem(<Bot>: W, <7>: Y)) <==> MergeMem(<Bot>: W, <7>: Y)
3687 //
3688 // But, in such a cascade, repeated memory slices can "block the leak":
3689 //   MergeMem(<Bot>: MergeMem(<Bot>: W, <7>: Y), <7>: Y') <==> MergeMem(<Bot>: W
3690 //
3691 // In the last example, Y is not part of the combined memory state of the
3692 // outermost MergeMem.  The system must, of course, prevent unschedulable
3693 // memory states from arising, so you can be sure that the state Y is somehow
3694 // a precursor to state Y'.
3695 //
3696 //
3697 // REPRESENTATION OF MEMORY MERGES: The indexes used to address the Node::in arr
3698 // of each MergeMemNode array are exactly the numerical alias indexes, including
3699 // but not limited to AliasIdxTop, AliasIdxBot, and AliasIdxRaw.  The functions
3700 // Compile::alias_type (and kin) produce and manage these indexes.
3701 //
3702 // By convention, the value of in(AliasIdxTop) (i.e., in(1)) is always the top n
3703 // (Note that this provides quick access to the top node inside MergeMem methods
3704 // without the need to reach out via TLS to Compile::current.)
3705 //
3706 // As a consequence of what was just described, a MergeMem that represents a ful
3707 // memory state has an edge in(AliasIdxBot) which is a "wide" memory state,
3708 // containing all alias categories.
3709 //
3710 // MergeMem nodes never (?) have control inputs, so in(0) is NULL.
3711 //
3712 // All other edges in(N) (including in(AliasIdxRaw), which is in(3)) are either
3713 // a memory state for the alias type <N>, or else the top node, meaning that
3714 // there is no particular input for that alias type.  Note that the length of
3715 // a MergeMem is variable, and may be extended at any time to accommodate new
3716 // memory states at larger alias indexes.  When merges grow, they are of course
3717 // filled with "top" in the unused in() positions.
3718 //
```

```
3719 // This use of top is named "empty_memory()", or "empty_mem" (no-memory) as a va
3720 // (Top was chosen because it works smoothly with passes like GCM.)
3721 //
3722 // For convenience, we hardwire the alias index for TypeRawPtr::BOTTOM.  (It is
3723 // the type of random VM bits like TLS references.)  Since it is always the
3724 // first non-Bot memory slice, some low-level loops use it to initialize an
3725 // index variable:  for (i = AliasIdxRaw; i < req(); i++).
3726 //
3727 //
3728 // ACCESSORS:  There is a special accessor MergeMemNode::base_memory which retur
3729 // the distinguished "wide" state.  The accessor MergeMemNode::memory_at(N) retu
3730 // the memory state for alias type <N>, or (if there is no particular slice at <
3731 // it returns the base memory.  To prevent bugs, memory_at does not accept <Top>
3732 // or <Bot> indexes.  The iterator MergeMemStream provides robust iteration over
3733 // MergeMem nodes or pairs of such nodes, ensuring that the non-top edges are vi
3734 //
3735 // %%%% We may get rid of base_memory as a separate accessor at some point; it i
3736 // really that different from the other memory inputs.  An abbreviation called
3737 // "bot_memory()" for "memory_at(AliasIdxBot)" would keep code tidy.
3738 //
3739 //
3740 // PARTIAL MEMORY STATES:  During optimization, MergeMem nodes may arise that re
3741 // partial memory states.  When a Phi splits through a MergeMem, the copy of the
3742 // that "emerges though" the base memory will be marked as excluding the alias t
3743 // of the other (narrow-memory) copies which "emerged through" the narrow edges:
3744 //
3745 //   Phi<Bot>(U, MergeMem(<Bot>: W, <8>: Y))
3746 //     ==Ideal=>  MergeMem(<Bot>: Phi<Bot-8>(U, W), Phi<8>(U, Y))
3747 //
3748 // This strange "subtraction" effect is necessary to ensure IGVN convergence.
3749 // (It is currently unimplemented.)  As you can see, the resulting merge is
3750 // actually a disjoint union of memory states, rather than an overlay.
3751 //

3753 //------------------------------MergeMemNode-----------------------------------
3754 Node* MergeMemNode::make_empty_memory() {
3755   Node* empty_memory = (Node*) Compile::current()->top();
3756   assert(empty_memory->is_top(), "correct sentinel identity");
3757   return empty_memory;
3758 }

3760 MergeMemNode::MergeMemNode(Node *new_base) : Node(1+Compile::AliasIdxRaw) {
3761   init_class_id(Class_MergeMem);
3762   // all inputs are nullified in Node::Node(int)
3763   // set_input(0, NULL);  // no control input

3765   // Initialize the edges uniformly to top, for starters.
3766   Node* empty_mem = make_empty_memory();
3767   for (uint i = Compile::AliasIdxTop; i < req(); i++) {
3768     init_req(i,empty_mem);
3769   }
3770   assert(empty_memory() == empty_mem, "");

3772   if( new_base != NULL && new_base->is_MergeMem() ) {
3773     MergeMemNode* mdef = new_base->as_MergeMem();
3774     assert(mdef->empty_memory() == empty_mem, "consistent sentinels");
3775     for (MergeMemStream mms(this, mdef); mms.next_non_empty2(); ) {
3776       mms.set_memory(mms.memory2());
3777     }
3778     assert(base_memory() == mdef->base_memory(), "");
3779   } else {
3780     set_base_memory(new_base);
3781   }
3782 }

3784 // Make a new, untransformed MergeMem with the same base as 'mem'.
```



```
3785 // If mem is itself a MergeMem, populate the result with the same edges.
3786 MergeMemNode* MergeMemNode::make(Compile* C, Node* mem) {
3787   return new(C, 1+Compile::AliasIdxRaw) MergeMemNode(mem);
3788 }

3790 //------------------------------cmp--------------------------------------------
3791 uint MergeMemNode::hash() const { return NO_HASH; }
3792 uint MergeMemNode::cmp( const Node &n ) const {
3793   return (&n == this);          // Always fail except on self
3794 }

3796 //------------------------------Identity---------------------------------------
3797 Node* MergeMemNode::Identity(PhaseTransform *phase) {
3798   // Identity if this merge point does not record any interesting memory
3799   // disambiguations.
3800   Node* base_mem = base_memory();
3801   Node* empty_mem = empty_memory();
3802   if (base_mem != empty_mem) {  // Memory path is not dead?
3803     for (uint i = Compile::AliasIdxRaw; i < req(); i++) {
3804       Node* mem = in(i);
3805       if (mem != empty_mem && mem != base_mem) {
3806         return this;            // Many memory splits; no change
3807       }
3808     }
3809   }
3810   return base_mem;              // No memory splits; ID on the one true input
3811 }

3813 //------------------------------Ideal------------------------------------------
3814 // This method is invoked recursively on chains of MergeMem nodes
3815 Node *MergeMemNode::Ideal(PhaseGVN *phase, bool can_reshape) {
3816   // Remove chain'd MergeMems
3817   //
3818   // This is delicate, because the each "in(i)" (i >= Raw) is interpreted
3819   // relative to the "in(Bot)".  Since we are patching both at the same time,
3820   // we have to be careful to read each "in(i)" relative to the old "in(Bot)",
3821   // but rewrite each "in(i)" relative to the new "in(Bot)".
3822   Node *progress = NULL;


3825   Node* old_base = base_memory();
3826   Node* empty_mem = empty_memory();
3827   if (old_base == empty_mem)
3828     return NULL; // Dead memory path.

3830   MergeMemNode* old_mbase;
3831   if (old_base != NULL && old_base->is_MergeMem())
3832     old_mbase = old_base->as_MergeMem();
3833   else
3834     old_mbase = NULL;
3835   Node* new_base = old_base;

3837   // simplify stacked MergeMems in base memory
3838   if (old_mbase)  new_base = old_mbase->base_memory();

3840   // the base memory might contribute new slices beyond my req()
3841   if (old_mbase)  grow_to_match(old_mbase);

3843   // Look carefully at the base node if it is a phi.
3844   PhiNode* phi_base;
3845   if (new_base != NULL && new_base->is_Phi())
3846     phi_base = new_base->as_Phi();
3847   else
3848     phi_base = NULL;

3850   Node*    phi_reg = NULL;
```

```
3851   uint     phi_len = (uint)-1;
3852   if (phi_base != NULL && !phi_base->is_copy()) {
3853     // do not examine phi if degraded to a copy
3854     phi_reg = phi_base->region();
3855     phi_len = phi_base->req();
3856     // see if the phi is unfinished
3857     for (uint i = 1; i < phi_len; i++) {
3858       if (phi_base->in(i) == NULL) {
3859         // incomplete phi; do not look at it yet!
3860         phi_reg = NULL;
3861         phi_len = (uint)-1;
3862         break;
3863       }
3864     }
3865   }

3867   // Note:  We do not call verify_sparse on entry, because inputs
3868   // can normalize to the base_memory via subsume_node or similar
3869   // mechanisms.  This method repairs that damage.

3871   assert(!old_mbase || old_mbase->is_empty_memory(empty_mem), "consistent sentin

3873   // Look at each slice.
3874   for (uint i = Compile::AliasIdxRaw; i < req(); i++) {
3875     Node* old_in = in(i);
3876     // calculate the old memory value
3877     Node* old_mem = old_in;
3878     if (old_mem == empty_mem)  old_mem = old_base;
3879     assert(old_mem == memory_at(i), "");

3881     // maybe update (reslice) the old memory value

3883     // simplify stacked MergeMems
3884     Node* new_mem = old_mem;
3885     MergeMemNode* old_mmem;
3886     if (old_mem != NULL && old_mem->is_MergeMem())
3887       old_mmem = old_mem->as_MergeMem();
3888     else
3889       old_mmem = NULL;
3890     if (old_mmem == this) {
3891       // This can happen if loops break up and safepoints disappear.
3892       // A merge of BotPtr (default) with a RawPtr memory derived from a
3893       // safepoint can be rewritten to a merge of the same BotPtr with
3894       // the BotPtr phi coming into the loop.  If that phi disappears
3895       // also, we can end up with a self-loop of the mergemem.
3896       // In general, if loops degenerate and memory effects disappear,
3897       // a mergemem can be left looking at itself.  This simply means
3898       // that the mergemem's default should be used, since there is
3899       // no longer any apparent effect on this slice.
3900       // Note: If a memory slice is a MergeMem cycle, it is unreachable
3901       //       from start.  Update the input to TOP.
3902       new_mem = (new_base == this || new_base == empty_mem)? empty_mem : new_bas
3903     }
3904     else if (old_mmem != NULL) {
3905       new_mem = old_mmem->memory_at(i);
3906     }
3907     // else preceding memory was not a MergeMem

3909     // replace equivalent phis (unfortunately, they do not GVN together)
3910     if (new_mem != NULL && new_mem != new_base &&
3911         new_mem->req() == phi_len && new_mem->in(0) == phi_reg) {
3912       if (new_mem->is_Phi()) {
3913         PhiNode* phi_mem = new_mem->as_Phi();
3914         for (uint i = 1; i < phi_len; i++) {
3915           if (phi_base->in(i) != phi_mem->in(i)) {
3916             phi_mem = NULL;
```



```
3917             break;
3918           }
3919         }
3920         if (phi_mem != NULL) {
3921           // equivalent phi nodes; revert to the def
3922           new_mem = new_base;
3923         }
3924       }
3925     }

3927     // maybe store down a new value
3928     Node* new_in = new_mem;
3929     if (new_in == new_base)  new_in = empty_mem;

3931     if (new_in != old_in) {
3932       // Warning:  Do not combine this "if" with the previous "if"
3933       // A memory slice might have be be rewritten even if it is semantically
3934       // unchanged, if the base_memory value has changed.
3935       set_req(i, new_in);
3936       progress = this;          // Report progress
3937     }
3938   }

3940   if (new_base != old_base) {
3941     set_req(Compile::AliasIdxBot, new_base);
3942     // Don't use set_base_memory(new_base), because we need to update du.
3943     assert(base_memory() == new_base, "");
3944     progress = this;
3945   }

3947   if( base_memory() == this ) {
3948     // a self cycle indicates this memory path is dead
3949     set_req(Compile::AliasIdxBot, empty_mem);
3950   }

3952   // Resolve external cycles by calling Ideal on a MergeMem base_memory
3953   // Recursion must occur after the self cycle check above
3954   if( base_memory()->is_MergeMem() ) {
3955     MergeMemNode *new_mbase = base_memory()->as_MergeMem();
3956     Node *m = phase->transform(new_mbase);  // Rollup any cycles
3957     if( m != NULL && (m->is_top() ||
3958         m->is_MergeMem() && m->as_MergeMem()->base_memory() == empty_mem) ) {
3959       // propagate rollup of dead cycle to self
3960       set_req(Compile::AliasIdxBot, empty_mem);
3961     }
3962   }

3964   if( base_memory() == empty_mem ) {
3965     progress = this;
3966     // Cut inputs during Parse phase only.
3967     // During Optimize phase a dead MergeMem node will be subsumed by Top.
3968     if( !can_reshape ) {
3969       for (uint i = Compile::AliasIdxRaw; i < req(); i++) {
3970         if( in(i) != empty_mem ) { set_req(i, empty_mem); }
3971       }
3972     }
3973   }

3975   if( !progress && base_memory()->is_Phi() && can_reshape ) {
3976     // Check if PhiNode::Ideal's "Split phis through memory merges"
3977     // transform should be attempted. Look for this->phi->this cycle.
3978     uint merge_width = req();
3979     if (merge_width > Compile::AliasIdxRaw) {
3980       PhiNode* phi = base_memory()->as_Phi();
3981       for( uint i = 1; i < phi->req(); ++i ) {// For all paths in
3982         if (phi->in(i) == this) {
```

```
3983           phase->is_IterGVN()->_worklist.push(phi);
3984           break;
3985         }
3986       }
3987     }
3988   }

3990   assert(progress || verify_sparse(), "please, no dups of base");
3991   return progress;
3992 }

3994 //------------------------set_base_memory-------------------------------------
3995 void MergeMemNode::set_base_memory(Node *new_base) {
3996   Node* empty_mem = empty_memory();
3997   set_req(Compile::AliasIdxBot, new_base);
3998   assert(memory_at(req()) == new_base, "must set default memory");
3999   // Clear out other occurrences of new_base:
4000   if (new_base != empty_mem) {
4001     for (uint i = Compile::AliasIdxRaw; i < req(); i++) {
4002       if (in(i) == new_base)  set_req(i, empty_mem);
4003     }
4004   }
4005 }

4007 //------------------------------out_RegMask-----------------------------------
4008 const RegMask &MergeMemNode::out_RegMask() const {
4009   return RegMask::Empty;
4010 }

4012 //------------------------------dump_spec-------------------------------------
4013 #ifndef PRODUCT
4014 void MergeMemNode::dump_spec(outputStream *st) const {
4015   st->print(" {");
4016   Node* base_mem = base_memory();
4017   for( uint i = Compile::AliasIdxRaw; i < req(); i++ ) {
4018     Node* mem = memory_at(i);
4019     if (mem == base_mem) { st->print(" -"); continue; }
4020     st->print( " N%d:", mem->_idx );
4021     Compile::current()->get_adr_type(i)->dump_on(st);
4022   }
4023   st->print(" }");
4024 }
4025 #endif // !PRODUCT


4028 #ifdef ASSERT
4029 static bool might_be_same(Node* a, Node* b) {
4030   if (a == b)  return true;
4031   if (!(a->is_Phi() || b->is_Phi()))  return false;
4032   // phis shift around during optimization
4033   return true;  // pretty stupid...
4034 }

4036 // verify a narrow slice (either incoming or outgoing)
4037 static void verify_memory_slice(const MergeMemNode* m, int alias_idx, Node* n) {
4038   if (!VerifyAliases)       return;  // don't bother to verify unless requested
4039   if (is_error_reported())  return;  // muzzle asserts when debugging an error
4040   if (Node::in_dump())      return;  // muzzle asserts when printing
4041   assert(alias_idx >= Compile::AliasIdxRaw, "must not disturb base_memory or sen
4042   assert(n != NULL, "");
4043   // Elide intervening MergeMem's
4044   while (n->is_MergeMem()) {
4045     n = n->as_MergeMem()->memory_at(alias_idx);
4046   }
4047   Compile* C = Compile::current();
4048   const TypePtr* n_adr_type = n->adr_type();
```



```
4049   if (n == m->empty_memory()) {
4050     // Implicit copy of base_memory()
4051   } else if (n_adr_type != TypePtr::BOTTOM) {
4052     assert(n_adr_type != NULL, "new memory must have a well-defined adr_type");
4053     assert(C->must_alias(n_adr_type, alias_idx), "new memory must match selected
4054   } else {
4055     // A few places like make_runtime_call "know" that VM calls are narrow,
4056     // and can be used to update only the VM bits stored as TypeRawPtr::BOTTOM.
4057     bool expected_wide_mem = false;
4058     if (n == m->base_memory()) {
4059       expected_wide_mem = true;
4060     } else if (alias_idx == Compile::AliasIdxRaw ||
4061                n == m->memory_at(Compile::AliasIdxRaw)) {
4062       expected_wide_mem = true;
4063     } else if (!C->alias_type(alias_idx)->is_rewritable()) {
4064       // memory can "leak through" calls on channels that
4065       // are write-once.  Allow this also.
4066       expected_wide_mem = true;
4067     }
4068     assert(expected_wide_mem, "expected narrow slice replacement");
4069   }
4070 }
4071 #else // !ASSERT
4072 #define verify_memory_slice(m,i,n) (0)  // PRODUCT version is no-op
4073 #endif


4076 //-----------------------------memory_at---------------------------------------
4077 Node* MergeMemNode::memory_at(uint alias_idx) const {
4078   assert(alias_idx >= Compile::AliasIdxRaw ||
4079          alias_idx == Compile::AliasIdxBot && Compile::current()->AliasLevel() =
4080          "must avoid base_memory and AliasIdxTop");

4082   // Otherwise, it is a narrow slice.
4083   Node* n = alias_idx < req() ? in(alias_idx) : empty_memory();
4084   Compile *C = Compile::current();
4085   if (is_empty_memory(n)) {
4086     // the array is sparse; empty slots are the "top" node
4087     n = base_memory();
4088     assert(Node::in_dump()
4089            || n == NULL || n->bottom_type() == Type::TOP
4090            || n->adr_type() == NULL // address is TOP
4091            || n->adr_type() == TypePtr::BOTTOM
4092            || n->adr_type() == TypeRawPtr::BOTTOM
4093            || Compile::current()->AliasLevel() == 0,
4094            "must be a wide memory");
4095     // AliasLevel == 0 if we are organizing the memory states manually.
4096     // See verify_memory_slice for comments on TypeRawPtr::BOTTOM.
4097   } else {
4098     // make sure the stored slice is sane
4099     #ifdef ASSERT
4100     if (is_error_reported() || Node::in_dump()) {
4101     } else if (might_be_same(n, base_memory())) {
4102       // Give it a pass:  It is a mostly harmless repetition of the base.
4103       // This can arise normally from node subsumption during optimization.
4104     } else {
4105       verify_memory_slice(this, alias_idx, n);
4106     }
4107     #endif
4108   }
4109   return n;
4110 }

4112 //--------------------------set_memory_at-------------------------------------
4113 void MergeMemNode::set_memory_at(uint alias_idx, Node *n) {
4114   verify_memory_slice(this, alias_idx, n);
```

```
4115   Node* empty_mem = empty_memory();
4116   if (n == base_memory())  n = empty_mem;  // collapse default
4117   uint need_req = alias_idx+1;
4118   if (req() < need_req) {
4119     if (n == empty_mem)  return;  // already the default, so do not grow me
4120     // grow the sparse array
4121     do {
4122       add_req(empty_mem);
4123     } while (req() < need_req);
4124   }
4125   set_req( alias_idx, n );
4126 }



4130 //--------------------------iteration_setup------------------------------------
4131 void MergeMemNode::iteration_setup(const MergeMemNode* other) {
4132   if (other != NULL) {
4133     grow_to_match(other);
4134     // invariant:  the finite support of mm2 is within mm->req()
4135     #ifdef ASSERT
4136     for (uint i = req(); i < other->req(); i++) {
4137       assert(other->is_empty_memory(other->in(i)), "slice left uncovered");
4138     }
4139     #endif
4140   }
4141   // Replace spurious copies of base_memory by top.
4142   Node* base_mem = base_memory();
4143   if (base_mem != NULL && !base_mem->is_top()) {
4144     for (uint i = Compile::AliasIdxBot+1, imax = req(); i < imax; i++) {
4145       if (in(i) == base_mem)
4146         set_req(i, empty_memory());
4147     }
4148   }
4149 }

4151 //--------------------------grow_to_match-------------------------------------
4152 void MergeMemNode::grow_to_match(const MergeMemNode* other) {
4153   Node* empty_mem = empty_memory();
4154   assert(other->is_empty_memory(empty_mem), "consistent sentinels");
4155   // look for the finite support of the other memory
4156   for (uint i = other->req(); --i >= req(); ) {
4157     if (other->in(i) != empty_mem) {
4158       uint new_len = i+1;
4159       while (req() < new_len)  add_req(empty_mem);
4160       break;
4161     }
4162   }
4163 }

4165 //---------------------------verify_sparse-------------------------------------
4166 #ifndef PRODUCT
4167 bool MergeMemNode::verify_sparse() const {
4168   assert(is_empty_memory(make_empty_memory()), "sane sentinel");
4169   Node* base_mem = base_memory();
4170   // The following can happen in degenerate cases, since empty==top.
4171   if (is_empty_memory(base_mem))  return true;
4172   for (uint i = Compile::AliasIdxRaw; i < req(); i++) {
4173     assert(in(i) != NULL, "sane slice");
4174     if (in(i) == base_mem)  return false;  // should have been the sentinel valu
4175   }
4176   return true;
4177 }

4179 bool MergeMemStream::match_memory(Node* mem, const MergeMemNode* mm, int idx) {
4180   Node* n;
```



```
4181   n = mm->in(idx);
4182   if (mem == n)  return true;  // might be empty_memory()
4183   n = (idx == Compile::AliasIdxBot)? mm->base_memory(): mm->memory_at(idx);
4184   if (mem == n)  return true;
4185   while (n->is_Phi() && (n = n->as_Phi()->is_copy()) != NULL) {
4186     if (mem == n)  return true;
4187     if (n == NULL)  break;
4188   }
4189   return false;
4190 }
4191 #endif // !PRODUCT
```

```
**********************************************************
   20541 Thu Sep  1 02:19:28 2011
new/src/share/vm/opto/parse3.cpp
**********************************************************
______unchanged_portion_omitted_


 147 void Parse::do_get_xxx(Node* obj, ciField* field, bool is_field) {
 148   // Does this field have a constant value?  If so, just push the value.
 149   if (field->is_constant()) {
 150     // final field
 151 #endif /* ! codereview */
 152     if (field->is_static()) {
 153       // final static field
 154       if (push_constant(field->constant_value()))
 155         return;
 156     }
 157     else {
 158       // final non-static field
 159       // Treat final non-static fields of trusted classes (classes in
 160       // java.lang.invoke and sun.invoke packages and subpackages) as
 161       // compile time constants.
 150       // final non-static field of a trusted class (classes in
 151       // java.lang.invoke and sun.invoke packages and subpackages).
 162       if (obj->is_Con()) {
 163         const TypeOopPtr* oop_ptr = obj->bottom_type()->isa_oopptr();
 164         ciObject* constant_oop = oop_ptr->const_oop();
 165         ciConstant constant = field->constant_value_of(constant_oop);

 166         if (push_constant(constant, true))
 167           return;
 168       }
 169     }
 170   }

 172   ciType* field_klass = field->type();
 173   bool is_vol = field->is_volatile();

 175   // Compute address and memory type.
 176   int offset = field->offset_in_bytes();
 177   const TypePtr* adr_type = C->alias_type(field)->adr_type();
 178   Node *adr = basic_plus_adr(obj, obj, offset);
 179   BasicType bt = field->layout_type();

 181   // Build the resultant type of the load
 182   const Type *type;

 184   bool must_assert_null = false;

 186   if( bt == T_OBJECT ) {
 187     if (!field->type()->is_loaded()) {
 188       type = TypeInstPtr::BOTTOM;
 189       must_assert_null = true;
 190     } else if (field->is_constant() && field->is_static()) {
 191       // This can happen if the constant oop is non-perm.
 192       ciObject* con = field->constant_value().as_object();
 193       // Do not "join" in the previous type; it doesn't add value,
 194       // and may yield a vacuous result if the field is of interface type.
 195       type = TypeOopPtr::make_from_constant(con)->isa_oopptr();
 196       assert(type != NULL, "field singleton type must be consistent");
 197     } else {
 198       type = TypeOopPtr::make_from_klass(field_klass->as_klass());
 199     }
 200   } else {
 201     type = Type::get_const_basic_type(bt);
 202   }
```



```
 203   // Build the load.
 204   Node* ld = make_load(NULL, adr, type, bt, adr_type, is_vol);

 206   // Adjust Java stack
 207   if (type2size[bt] == 1)
 208     push(ld);
 209   else
 210     push_pair(ld);

 212   if (must_assert_null) {
 213     // Do not take a trap here.  It's possible that the program
 214     // will never load the field's class, and will happily see
 215     // null values in this field forever.  Don't stumble into a
 216     // trap for such a program, or we might get a long series
 217     // of useless recompilations.  (Or, we might load a class
 218     // which should not be loaded.)  If we ever see a non-null
 219     // value, we will then trap and recompile.  (The trap will
 220     // not need to mention the class index, since the class will
 221     // already have been loaded if we ever see a non-null value.)
 222     // uncommon_trap(iter().get_field_signature_index());
 223 #ifndef PRODUCT
 224     if (PrintOpto && (Verbose || WizardMode)) {
 225       method()->print_name(); tty->print_cr(" asserting nullness of field at bci
 226     }
 227 #endif
 228     if (C->log() != NULL) {
 229       C->log()->elem("assert_null reason='field' klass='%d'",
 230                      C->log()->identify(field->type()));
 231     }
 232     // If there is going to be a trap, put it at the next bytecode:
 233     set_bci(iter().next_bci());
 234     do_null_assert(peek(), T_OBJECT);
 235     set_bci(iter().cur_bci()); // put it back
 236   }

 238   // If reference is volatile, prevent following memory ops from
 239   // floating up past the volatile read.  Also prevents commoning
 240   // another volatile read.
 241   if (field->is_volatile()) {
 242     // Memory barrier includes bogus read of value to force load BEFORE membar
 243     insert_mem_bar(Op_MemBarAcquire, ld);
 244   }
 245 }
______unchanged_portion_omitted_
```