C2 compiler extensions and new optimizations

The PPC port adds some minor extensions to the C2 compiler to support the PPC architecture. We also implemented some new optimizations.

Extend adl/adlc by effect TEMP_DEF.

Adl is the language used to express code generation patterns in the C2 compiler of hotspot. Effects are used to specify additional properties of IR nodes that can not be derived from the match rules. E.g., the USE_DEF effect allows to specify that a register used by a node and a register defined by the node must be identical, as required by two-address assembler instructions.

The new TEMP_DEF effect is similar to USE_DEF, except that a TEMP node will be generated that represents the USE.

With this effect one can express that the def'ed register must be different from the used ones corresponding to ins. Currently this is already possible by specifying effect TEMP for the operand with effect DEF from the match rule.

Introducing this new identifyer makes the code more readable and allows to specify the effect for nodes without match rules.

An example is an optimized encode node, if the base of the compressed heap is 35G aligned, i.e., the shifted narrow oop can be merged with the base by an or instruction.

On PPC we can shift and or with a single instruction, so we can implement Decode with these instructions:

```
mov Rdst = Rbase
rldimi Rdst = Rdst || (Rsrc << 3)
```

As the move is off the critical path, this is superior to do a shift and an add. Unfortunately we must guarantee that Rdst != Rsrc. which we do with a TEMP_DEF effect:

```
instruct decodeN(iRegPdst dst, iRegNsrc src) %{
    match(Set dst (DecodeN src));
    effect(TEMP_DEF dst);
```

ImplicitNullChecks on operating systems where the zero page is not read protected.

The C2 compiler uses load or store operations with sufficiently small offsets to do null checks. It risks that the protected page at address zero is hit, and returns via the signal handler to handle the Java NullPointerException. The code assumes that the zero page is read and write protected. Alternatively, the optimization can be switched off altogether. ImplicitNullChecks are a very profitable optimizations On Aix, the zero page is only write protected, thus ImplicitNullChecks as described above can not be used. We extended the optimization to work on Aix.

We introduce a property zero_page_read_protected in the os layer. This is set to false on Aix, to true on others. The ImplicitNullCheck optimization only considers Stores as operations to check for NULL if this property is set.

With compressed oops, a special protected page is placed before the heap so that a decoded NULL points into this page and an access traps. The protection of this page is independent of the protection of the zero page. Our fix considers this so that for heap-based compressed oops also Loads are considered as null check operations.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/2c8a195051eb

Trap based null and range checks

PPC offers an instruction that does a compare-and-trap. I.e., it throws a SIGTRAP if the comparison failed. This instruction is very cheap if the comparison is true.

We use this instruction to do NullChecks where ImplicitNullChecks are not applicable. Especially on Aix, where the zero page is not read protected, these are many. Further we do range checks with this instruction.

We introduce these instructions during matching if the probability of the branch to be taken is very low. Unfortunately this requires changing shared code, as the block layout algorithm and construction of the ... are not aware of these nodes.

We introduce two new flags TrapBasedNullChecks and TrapBasedRangeChecks and a function Node::is_TrapBasedCheckNode(). We adapt FillExceptionTables() and PhaseCFG::fixup_flow() to handle these nodes. In fixup_flow() we take care that the case that traps is the taken branch.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/f9ce6652d9c2

This change also requires that the linux os implementation supports SIGTRAP which is introduced here:
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/5792f69f1881

Adlc: Fields in MachNodes.

We extend adl by a specification of fields and adlc to generate these.

If an instruct specification contains a line ins_<name>(<val>), adlc generates a function ins_<name>() returning the constant <val>.

We extended adlc to generate a field in the node if the declaration is ins_field_<name>(<type>). The field declaration is <type> _<name>;
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/d1dfaa602f1a

Adlc: Specify which nodes to rematerialize.

Adlc has a row of internal heuristics that determines which nodes should be rematerialized during register allocation. This is not very well suited for our port. Therefore we introduce to adl annotations ins_cannot_rematerialize() and ins_should_rematerialize() that complement this heuristic.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/eabde81a086d

Handling ordered loads and stores

The C2 compiler represents Java loads and stores that require memory ordering as a load/store node and a memory barrrier node. If necessary the memory barrier node emits an assembler operation establishing the desired memory ordering. Further the compiler does not reorder memory operations wrt. this operation.

On IA64, there are special memory operations for acquiring loads and releasing stores. On PPC, there is a special instructions sequence implementing an effective load acquire that must know about the registers used in the load.

To support these operations, we extend the ideal load and store nodes to know whether they must acquire or release. They now have a flag set if they should acquire/release All graph kit methods have an additional argument setting the corresponding field.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/fb70eef44b05

On PPC, we only use this extension for loads, but we propose to also add this change for stores to OpenJDK for symmetry.

The C2 compiler does not know that it should not reorder wrt. to memory operations with the acquire/release flag set. Therefore we still issue the corresponding MemBar instructions after the memory operations, but they do not generate any assembly.

Unfortunately this raises problems if the MemBar nodes are used without corresponding loads, as in LibraryCallKit::inline_unsafe_fence().
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/bc97850c4977

To overcome this problem, we propose to generate the MemBar in the load/store factory methods if required. This could be omitted locally on PPC and IA64 or other platforms that emit the ordering operation along with the load. Instead, a MemBarCPUOrder could be emitted. Then, on these platforms, we can properly emit ordering operations for the MemBar nodes.

Trampoline stubs

On PPC relative branches and calls can only encode a 16-bit offset, i.e., they reach only 64k far. This is not sufficient for calls to reach all code, thus a long branch is needed. As loading 64-bit constants is expensive (5 instructions), and as patching them atomic is hard, we implement a long branch by loading the callee address from the constant pool. When a call instruction is relocated, we decide which branch to use.

Performance measurements showed that it is better to keep the long call in a stub. Also, this makes it easier to find a good instruction schedule. Changing the instruction sequence after fixing the schedule, as it happens with relocation, can break an instruction schedule for Power6 considerably. The Power6 processor derives a lot of information about implicit bundling from the instructions executed previously.

To implement this feature, we need a new relocation type trampoline_stub_Relocation.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/d02f0701be17

Constants, Constant pool and Calls.

On PPC, loading a coastant into a register requires five instructions. These instructions can't easily be patched atomically. Therefore we choose to load all 64-bit constants from the constant pool. Unfortunately the IC cache and the call target of Call nodes are not represented by Const nodes in the IR.

One can not use the adl constant pool functionality with Calls. Using $constanttablebase etc. with a call has two problems: It makes the call a subnode of MachConstantNode, but calls must be subnodes of SafePointNode. Further it adds an edge to the Call node containing the constant table base constant. But call nodes have a fixed layout of in-edges, thus this is not possible either.

Therefore we do not use $constanttablebase for Calls. We add the edge to the table base in an extra phase after matching. This phase walks the IR and fixes the Call nodes. We use the TypeFunc::ReturnAdr in-edge of Calls, which is not used on PPC, for this. This is an existing edge and thus does not break the other phases relying on the edge layout of Calls. To recognize nodes that need the constant table base input in this phase, we added a funcion ins_requires_toc() to MachNodes, which returns false by default, and true if specified in the adl instruct statements.

As Calls are not derived from MachConstantNode the phase computing the size of the constant pools skips these. To fix this we extend MachNodes by a function ins_num_consts() returning the space required in the constant pool by the node, and adapt the phase computing the space requirements.

We also use this functionality in the storeCM node, for which we implemented an optimization that requires a constant.

This change extends HotSpot by the new platform-dependent phase after matching:
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/45f271751014
An this one adds the PPC extensions for constants:
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/9a8d8eff3f61

Comment:

This change is designed to do as few functional changes to shared code as possible. Adding a new, but unused phase to sparc and x86 should do no harm there. But because of this it is not a really clean solution of the problem. We would be happy to redesign Call nodes so they can get new edges added.

Also, we could redesign the $constanttablebase functionality to support more constants similarly (polling page, narrow oop base). Further, it would be helpful if it would add a MachOper describing the new input edge. If a constant is recognize by having this operand, or by a function generated into the node, the problem with the ambiguous superclasses can be resolved.

Expanding nodes after register allocation (8003854).

We designed a compiler phase the expands IR nodes after register allocation. We call this phase lateExpand.

Some nodes can not be expanded during matching. E.g., register allocation might not be able to deal with the resulting pattern, or global code motion might break some constraints. But instruction scheduling needs to be able to place each instruction individiually, thus a node should correspond to a single instruction if possible. The lateExpand phase which runs after register allocation solves this. Whether and how nodes are expanded is specified in the ad-file. Shared code calls the exand routines if they are available. We use this for some nodes on ppc, and extensively on ia64.

LateExpand is called after register allocation, just before output (i.e. scheduling). It only gets called if Matcher::require_late_expand is true. It expands compound nodes requiring several assembler instructions to be implemented into two or more non-compound nodes. The old compound node is simply replaced in its location in the basic block by a new subgraph which does not contain compound nodes any more. The scheduler called during output can process these non-compound nodes.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/19846affb789

Implementation details

Nodes requiring late expand are specified in the ad file by using an lateExpand statement instead of ins_encode. A lateExpand contains a single call to an encoding, as does an ins_encode statement. Instead of an emit() emit function a lateExpand() function is generated that doesn't emit assembler but creates a new subgraph. A walk over the IR calls this lateExpand function for each node that specifies a lateExpand. This function returns the new nodes generated in an array passed in the call. The old node, potential MachTemps before and potential Projs after it then get disconnected and replaced by the new nodes. The instruction generating the result has to be the last one in the array. In general it is assumed that Projs after the node expanded are kills. These kills are not required any more after expanding as there are now explicitly visible def-use chains and the Projs are removed. This does not hold for calls: They do not only have kill-Projs but also Projs defining values. Therefore Projs after the node expanded are removed for all but for calls. If a node is to be reused, it must be added to the nodes list returned, and it will be added again.

Implementing the lateExpand function for a node is rather tedious. It requires knowledge about many node details, as the nodes and the subgraph must be hand crafted. To simplify this, adlc generates some utility variables into the lateExpand function, e.g., holding the operands as specified by the lateExpand encoding specification, e.g.:

- unsigned idx_<par_name> holding the index of the node in the ins
- Node *n_<par_name>   holding the node loaded from the ins
- MachOpnd *op_<par_name> holding the corresponding operand

The ordering of operands can not be determined by looking at a rule. Especially if a match rule matches several different trees, several nodes are generated from one instruct specification with different operand orderings. In this case the adlc generated variables are the only way to access the ins and operands deterministically.

Example

Below you find an example how to use late expand for the sparc.ad file. Further down you see the code generated by adlc. Perhaps you can find better use cases for this feature.

```
--- a/src/cpu/sparc/vm/sparc.ad   2012-11-21 12:27:04.591486000 +0100
+++ b/src/cpu/sparc/vm/sparc.ad    2012-11-19 14:45:15.059452000 +0100
@@ -1933,7 +1937,7 @@
 }

  // Does the CPU require late expand (see block.cpp for description of late expand)?
-const bool Matcher::require_late_expand = false;
+const bool Matcher::require_late_expand = true;

  // Should the Matcher clone shifts on addressing modes, expecting them to
  // be subsumed into complex addressing expressions or compute them into
@@ -7497,6 +7501,7 @@
  // Register Division
  instruct divI_reg_reg(iRegI dst, iRegIsafe src1, iRegIsafe src2) %{
    match(Set dst (DivI src1 src2));
+  predicate(!UseNewCode);
    ins_cost((2+71)*DEFAULT_COST);

    format %{ "SRA     $src2,0,$src2\n\t"
@@ -7506,6 +7511,68 @@
    ins_pipe(sdiv_reg_reg);
  %}
```

```
+//----------------------------------------------------------------------------------
+
+encode %{
+
+  enc_class lateExpandIdiv_reg_reg(iRegI dst, iRegIsafe src1, iRegIsafe src2) %{
+    MachNode *m1 = new (C) divI_reg_reg_SRANode();
+    MachNode *m2 = new (C) divI_reg_reg_SRANode();
+    MachNode *m3 = new (C) divI_reg_reg_SDIVXNode();
+
+    m1->add_req(n_region, n_src1);
+    m2->add_req(n_region, n_src2);
+    m3->add_req(n_region, m1, m2);
+
+    m1->_opnds[0] = _opnds[1]->clone(C);
+    m1->_opnds[1] = _opnds[1]->clone(C);
+
+    m2->_opnds[0] = _opnds[2]->clone(C);
+    m2->_opnds[1] = _opnds[2]->clone(C);
+
+    m3->_opnds[0] = _opnds[0]->clone(C);
+    m3->_opnds[1] = _opnds[1]->clone(C);
+    m3->_opnds[2] = _opnds[2]->clone(C);
+
+    ra_->set1(m1->_idx, ra_->get_reg_first(n_src1));
+    ra_->set1(m2->_idx, ra_->get_reg_first(n_src2));
+    ra_->set1(m3->_idx, ra_->get_reg_first(this));
+
+    nodes->push(m1);
+    nodes->push(m2);
+    nodes->push(m3);
+  %}
+%}
+
+instruct divI_reg_reg_SRA(iRegIsafe dst) %{
+  effect(USE_DEF dst);
+  size(4);
+  format %{ "SRA     $dst,0,$dst\n\t" %}
+  ins_encode %{ __ sra($dst$$Register, 0, $dst$$Register); %}
+  ins_pipe(ialu_reg_reg);
+%}
+
+instruct divI_reg_reg_SDIVX(iRegI dst, iRegIsafe src1, iRegIsafe src2) %{
+  effect(DEF dst, USE src1, USE src2);
+  size(4);
+  format %{ "SDIVX   $src1,$src2,$dst\n\t" %}
+  ins_encode %{ __ sdivx($dst$$Register, 0, $dst$$Register); %}
+  ins_pipe(sdiv_reg_reg);
+%}
+
+instruct divI_reg_reg_Ex(iRegI dst, iRegIsafe src1, iRegIsafe src2) %{
+  match(Set dst (DivI src1 src2));
+  predicate(UseNewCode);
+  ins_cost((2+71)*DEFAULT_COST);
+
+  format %{ "SRA     $src2,0,$src2\n\t"
+            "SRA     $src1,0,$src1\n\t"
+            "SDIVX   $src1,$src2,$dst" %}
+  lateExpand( lateExpandIdiv_reg_reg(src1, src2, dst) );
+%}
+
+//----------------------------------------------------------------------------------
+
 // Immediate Division
 instruct divI_reg_imm13(iRegI dst, iRegIsafe src1, immI13 src2) %{
   match(Set dst (DivI src1 src2));
```

Code generated by adlc:

```
class divI_reg_reg_ExNode : public MachNode {
  // ...
  virtual bool           requires_late_expand() const { return true; }
  virtual void           lateExpand(GrowableArray <Node *> *nodes, PhaseRegAlloc *ra_);
  // ...
};
```

```
void  divI_reg_reg_ExNode::lateExpand(GrowableArray <Node *> *nodes, PhaseRegAlloc *ra_) {
  // Start at oper_input_base() and count operands
  unsigned idx0 = 1;
  unsigned idx1 = 1;  // src1
  unsigned idx2 = idx1 + opnd_array(1)->num_edges();  // src2
  // Access to ins and operands for late expand.
  unsigned idx_dst   = idx1;  // iRegI,  src1
  unsigned idx_src1  = idx2;  // iRegIsafe,  src2
  unsigned idx_src2  = idx0;  // iRegIsafe,  dst
  Node    *n_region = lookup(0);
  Node    *n_dst    = lookup(idx_dst);
  Node    *n_src1   = lookup(idx_src1);
  Node    *n_src2   = lookup(idx_src2);
  iRegIOper *op_dst = (iRegIOper *)opnd_array(1);
  iRegIsafeOper *op_src1 = (iRegIsafeOper *)opnd_array(2);
  iRegIsafeOper *op_src2 = (iRegIsafeOper *)opnd_array(0);
  Compile *C = Compile::current();
  {
#line 7518 "/net/usr.work/d045726/oJ/8/main-hotspot-outputStream-test/src/cpu/sparc/vm/sparc.ad"

    MachNode *m1 = new (C) divI_reg_reg_SRANode();
    MachNode *m2 = new (C) divI_reg_reg_SRANode();
    MachNode *m3 = new (C) divI_reg_reg_SDIVXNode();

    m1->add_req(n_region, n_src1);
    m2->add_req(n_region, n_src2);
    m3->add_req(n_region, m1, m2);

    m1->_opnds[0] = _opnds[1]->clone(C);
    m1->_opnds[1] = _opnds[1]->clone(C);

    m2->_opnds[0] = _opnds[2]->clone(C);
    m2->_opnds[1] = _opnds[2]->clone(C);

    m3->_opnds[0] = _opnds[0]->clone(C);
    m3->_opnds[1] = _opnds[1]->clone(C);
    m3->_opnds[2] = _opnds[2]->clone(C);

    ra_->set1(m1->_idx, ra_->get_reg_first(n_src1));
    ra_->set1(m2->_idx, ra_->get_reg_first(n_src2));
    ra_->set1(m3->_idx, ra_->get_reg_first(this));

    nodes->push(m1);
    nodes->push(m2);
    nodes->push(m3);

#line 11120 "../generated/adfiles/ad_sparc.cpp"
  }
}
```

Trap based null checks

Trampoline relocations

etc.