[C2 X64 Overview]

With AVX-512 extension Intel architectures added 8 new opmask registers (K0-K7).

These are 64 bit registers used for predicated vector operations, where each

bit in opmask register corresponds to a vector lane element.

Currently a masked vector operation is expanded into a vector operation over entire

vector followed by a blend operation which selects between old and updated value of

destination vector based on the mask value present in the vector register.

On non-AVX512 targets a blend instruction performs the selection between source vector

lanes based on the MSB of the corresponding mask lane. AVX-512 blend instruction

performs selection based on the bit patterns in embedded opmask register, in order to

populate an opmask register an additional vector comparison operation is performed.

e.g.

  Java Snippet:

  -----

  IntVector  vec  = IntVector.fromArray(IntVector.SPECIES\_512, int\_arr, 0);

  VectorMask mask = VectorMask.fromArray(IntVector.SPECIES\_512, mask\_arr, 0);

  IntVector   res = vec1.lanewise(VectorOperations.ABS, mask);

  Existing JIT sequence:

  -----

  0x00007f792517309f:   vmovdqu32 0x10(%r9),%zmm0           // vec      = LoadVector int\_arr

  0x00007f79251730a9:   vmovdqu 0x10(%r12,%r8,8),%xmm1      // mask\_vec = LoadVector mask\_arr

  0x00007f79251730b0:   vpabsd %zmm0,%zmm2                  // ABS operation on entire vector.

  0x00007f79251730bc:   vpxord %zmm3,%zmm3,%zmm3            // Opmask register population through

  0x00007f79251730c2:   vpsubb %zmm1,%zmm3,%zmm3            // an explicit vector comparison operation.

  0x00007f79251730c8:   vpmovsxbd %xmm3,%zmm3               //

  0x00007f79251730ce:   vpcmpeqd -0xeb539(%rip),%zmm3,%k7   // mask = VectorLoadMask mask\_vec

  0x00007f79251730d9:   vpblendmd %zmm2,%zmm0,%zmm0{%k7}    // Vector blending operation.

  Proposed JIT sequence:

  -----

  0x00007f4a4c70559f:   vmovdqu32 0x10(%r9),%zmm1            // vec      = LoadVector int\_arr

  0x00007f4a4c7055a9:   vmovdqu 0x10(%r12,%r8,8),%xmm0       // mask\_vec = LoadVector mask\_arr

  0x00007f4a4c7055b6:   vpcmpb $0x0,-0xee9e1(%rip),%xmm0,%k7 // k7 = VectorLoadMask mask\_vec

  0x00007f4a4c7055c1:   vpabsd %zmm1,%zmm1{%k7}              // AVX-512 predicated  vector ABS operation.

As can be seen in above example using a predicated vector instruction which directly consumes

an opmask operand we can save an extra vector comparison and a vector blend instruction.

This improved both the emitted code size and latency for the masked operation.

X64 side support for masked operation optimization is divided into two stages.

1. Enhancing C2 register allocator to support allocation of opmask registers.
2. Improving the existing masked operation support.

Constraint:

-----

1. Non-AVX512 targets expects mask to be propagated through a vector of same shape as the

             Source operands of vector operation, whereas targets supporting AVX-512 feature

             expects mask to be present in 64 bit opmask register.

1. Curtail the number of new instruction selection patterns as ADLC process each pattern

and generates C++ code consumed by different phases of compiler backend, this effectively

adds to libjvm binary size.

[C2 generic modifications]

Currently C2 compiler does not have dedicated IR nodes for masked operations, also since masks

are propagated through vectors whose shape must match the shape of the source operands of a vector

operation hence mask generating nodes uses existing concrete vector types (TypeVect[SDXYZ]).

One of the design goal is to improve existing masked operation support with minimal impact on

IR and re-use existing IR mask generating nodes where ever possible and appropriate.

As per the constraints imposed by X64 ISA a mask generating node should have a different type

for targets supporting predicate registers.

Following changes are proposed in C2 compile:

1) Creation of a new dedicated concrete type (TypeVectMask) for mask generating nodes.

   Mask generating nodes (VectorLoadMask, VectorMaskGen , VectorMaskCmp , VectorMask.maskAll())

   should select between a usual vector type or a new mask type based on the existence of predicate

   register on target.

2) Creation of following a new IR node[s] for masked operations, this will carry the vector operation

   Information as additional meta data, this meta data will be propagated from ideal node to machine

   Node during instruction selection.

***VectorMaskedOper mask src1 [src2] [src3]***

   New IR node can be shared for different kinds of vector operations i.e. unary/binary and ternary or

   we can have a dedicated masked operation IR node one for each kind.

3) Ideal transformation to fold following graph pattern

***VectorBlend dst (VectorOperation dst src1 src2 src3) mask -> VectorMaskOper dst mask src1 [src2] [src3]***

   There are currently two routes though with vector nodes are created, one though direct use of Vector API

   intrinsic, here vector node are created during parsing. Secondly, when loops are auto vectorized some of the

   scalar operation can be replaced by vector operation.

   By performing above transformation during VectorBlend idealization will allow creation of masked operation

   nodes in both the scenarios.

4) Creation of new MachMetaData node, currently during machine node generation type information is propagated

   from ideal node to MachTypeNode, new MachMetaData node will inherit MachTypeNode and will carry

   any addition information passed from Ideal node to machine node. New VectorMaskedOper nodes will

   propagate vector operation information as meta data to MachMetaData node.

5) Instructions selection patterns for unary, binary and ternary masked operations.

   and ADLC changes to inherit MachNodes corresponding to new instruction patterns from MachMetaData class.

Apart from above changes, C2 register allocator needs to be extended to support predicated registers. This shall

involve following changes:-

1. ADLC changes: New register definitions, register classes, allocation classes, operand definitions and

spill code handling for opmask registers.

2)   Runtime: Save/restoration for opmask registers for 32 and 64 bit JVM.  
      a) For 64 bit JVM we are already reserving the space in the frame layout which comply with XSAVE layout,

          but are not saving and restoring at designated offset (1088). Hence no extra space overhead apart from

          save/restore cost.  
      b) For 32 bit JVM: Additional 64 byte are allocated apart from FXSTORE area on the lines of storage for ZMM(16-31)

          and YMM-Hi bank.

1. Replace all the hard-coded opmask references from macro-assembly routines, pull out

the opmask occurrences all the way up to instruction pattern and adding an unbounded opmask

operand for them. This exposes these operands to RA and scheduler and will automatically facilitate

any spilling of live opmask registers across call sites for any non-temporary opmask operand.

1. Register class initializations related to Op\_RegVMask during matcher startup. Enable opmask register spilling

To stack location and other opmask registers.