jminusminus

Class NInterval

- java.lang.Object
  - jminusminus.NInterval

- All Implemented Interfaces:
  Comparable<NInterval>

```
class NInterval
extends Object
implements Comparable<NInterval>
```

A lifetime interval, recording the interval of LIR code for which the corresponding virtual register contains a useful value.

- **Field Summary**

Fields

Modifier and Type	Field and Description
ArrayList<NInterval>	**children** Children of this interval.
int	**offset** Offset.
OffsetFrom	**offsetFrom** From offset.
NInterval	**parent** Parent of this interval.
NPhysicalRegister	**pRegister** The NPhyicalRegister assigned to this interval.

ArrayList<NRange>	**ranges** All live ranges for this virtual register
boolean	**spill** Whether or not to spill.
TreeMap<Integer,InstructionType>	**usePositions** All use positions (in LIR) and their types for this virtual register
int	**vRegId** The virtual register id corresponding to the index in the array list of NIntervals used by register allocation

- **Constructor Summary**

Constructors

Constructor and Description

NInterval(int virtualRegID, NControlFlowGraph cfg) Construct a NInterval with the given virtual register ID for the given control flow graph.
NInterval(int virtualRegID, NControlFlowGraph cfg, ArrayList<NRange> childRanges, NInterval parent) This second constructor is used in instantiating children of a split interval.

- **Method Summary**

Methods

Modifier and Type	Method and Description
void	**addOrExtendNRange**(NRange newNRange) Add a new range to the existing ranges.
void	**addUsePosition**(Integer index, InstructionType type) Register a use (read or write)>
NInterval	**childAt**(int idx) The child interval at a given instruction index.
NInterval	**childAtOrEndingBefore**(NBasicBlock b) A child of this interval which is live or ends before the given basic block's end.
NInterval	**childAtOrStartingAfter**(NBasicBlock b) The child of this interval which is live or starts after the given basic block's start
int	**compareTo**(NInterval other) Compare start positions (for ordering intervals).
int	**endsAtBlock**() The basic block in which this interval's end position falls.
boolean	**equals**(NInterval other) Two intervals are equal if they have the same virtual register

	ID.
int	**firstRangeStart**() The start position for the first range.
int	**firstUsage**() The first use position in this interval.
boolean	**isChild**() Is this interval a child interval?
boolean	**isLiveAt**(int atIndex) Check if this vreg is alive at a given index.
boolean	**isParent**() Is this interval a parent interval? (Ie, does it have children?)
int	**lastNRangeStop**() The stop position for the last range.
void	**newFirstRangeStart**(int newStart) Sets the start value of the very first range.
int	**nextIntersection**(NInterval otherInterval) Looks for the very first position where an intersection with another interval occurs.
int	**nextUseAfterStart**(NInterval currInterval) The next use position of this interval after the first range start of the foreign interval.
void	**spill**() Assigns an offset to this interval (if one hasn't been already assigned).
NInterval	**splitAt**(int idx) Split the current interval at the given index.
int	**startsAtBlock**() Returns the basic block in which this interval's start position falls.
void	**writeToStdOut**(PrettyPrinter p) Write the interval information to STDOUT.

- **Methods inherited from class java.lang.Object**

  `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

- **Field Detail**

  - **vRegId**

```
publicint vRegId
```

The virtual register id corresponding to the index in the array list of NIntervals used by register allocation

  - **ranges**

```
publicArrayList<NRange> ranges
```

All live ranges for this virtual register

- **usePositions**

```
publicTreeMap<Integer,InstructionType> usePositions
```

All use positions (in LIR) and their types for this virtual register

- **pRegister**

```
publicNPhysicalRegister pRegister
```

The NPhyicalRegister assigned to this interval. If an interval ends up needing more than one physical register it is split.

- **spill**

```
publicboolean spill
```

Whether or not to spill.

- **offsetFrom**

```
publicOffsetFrom offsetFrom
```

From offset.

- **offset**

```
publicint offset
```

Offset.

- **parent**

```
publicNInterval parent
```

Parent of this interval.

- **children**

```
publicArrayList<NInterval> children
```

Children of this interval.

- **Constructor Detail**

  - **NInterval**

```
publicNInterval(intvirtualRegID,
        NControlFlowGraphcfg)
```

Construct a NInterval with the given virtual register ID for the given control flow graph.

Parameters:

`virtualRegID` - program counter.
`cfg` - The control flow graph.

  - **NInterval**

```
publicNInterval(intvirtualRegID,
        NControlFlowGraphcfg,
        ArrayList<NRange>childRanges,
        NIntervalparent)
```

This second constructor is used in instantiating children of a split interval.

Parameters:

`virtualRegID` - program counter.
`cfg` - The control flow graph.
`childRanges` - The instruction ranges for this child.
`parent` - The parent interval.

- ## Method Detail

  - ### addOrExtendNRange

```
publicvoidaddOrExtendNRange(NRangenewNRange)
```

Add a new range to the existing ranges.

Parameters:

`newNRange` - - the NRange to add

  - ### nextIntersection

```
publicintnextIntersection(NIntervalotherInterval)
```

Looks for the very first position where an intersection with another interval occurs. NOTE: A.nextIntersection(B) equals B.nextIntersection(A)

Parameters:

`otherInterval` - the interval to compare against for intersection.

Returns:

the position where the intersection begins.

  - ### nextUsageOverlapping

```
publicintnextUsageOverlapping(NIntervalcurrInterval)
```


The next use position of this interval after the first range start of the foreign interval. If there is no such use, then the first use position is returned to preserve data flow (in case of loops).

Parameters:

`currInterval` - the interval with starting point after which we want to find the next usage of this one.

Returns:

the next use position.

  - ### firstUsage

```
publicintfirstUsage()
```

The first use position in this interval.

Returns:

the first use position.

  - ### newFirstRangeStart

```
publicvoidnewFirstRangeStart(intnewStart)
```

Sets the start value of the very first range. Note: There will always be at least one range before this method is used by buildIntervals.

Parameters:

`newStart` - the value to which the first range's start will be set.

  - ### addUsePosition

```
publicvoidaddUsePosition(Integerindex,
                         InstructionTypetype)
```

Register a use (read or write)>

Parameters:

`index` - the site of the use.

`type` - the instruction type.

- **isLiveAt**

```
publicbooleanisLiveAt(intatIndex)
```

Check if this vreg is alive at a given index.

Parameters:

`atIndex` - the index at which to see if this register is live.

- **writeToStdOut**

```
publicvoidwriteToStdOut(PrettyPrinterp)
```

Write the interval information to STDOUT.

Parameters:

p - for pretty printing with indentation.

- **firstRangeStart**

```
publicintfirstRangeStart()
```

The start position for the first range.

Returns:

the start position.

- **lastNRangeStop**

```
publicintlastNRangeStop()
```


The stop position for the last range.

Returns:

the stop position.

- **compareTo**

```
publicintcompareTo(NIntervalother)
```

Compare start positions (for ordering intervals).

Specified by:

`compareTo` in interface `Comparable<NInterval>`

Parameters:

`other` - interval to compare to.

Returns:

ordering value.

- **equals**

```
publicbooleanequals(NIntervalother)
```

Two intervals are equal if they have the same virtual register ID.

Parameters:

`other` - the interval we are comparing ourself with.

Returns:

true if the two intervals are the same, false otherwise.

- **splitAt**

```
publicNIntervalsplitAt(intidx)
```

Split the current interval at the given index. Responsible for splitting a range if the index falls on one, moving remaining ranges over to child, and moving appropriate usePositions over to the child.

Parameters:

`idx` - the index at which this interval is to be split

Returns:

the child interval which is to be sorted onto unhandled. If there was no child created in the case of a pruning this interval is returned.

- **childAt**

```
publicNIntervalchildAt(intidx)
```

The child interval at a given instruction index.

Parameters:

`idx` - The instruction index.

Returns:

the child interval.

- **childAtOrEndingBefore**

```
publicNIntervalchildAtOrEndingBefore(NBasicBlockb)
```

A child of this interval which is live or ends before the given basic block's end.

Parameters:

b - the basic block.

Returns:

the child of this interval which ends at or nearest (before) this basic block's end (last lir instruction index).

- **childAtOrStartingAfter**

```
publicNIntervalchildAtOrStartingAfter(NBasicBlockb)
```

The child of this interval which is live or starts after the given basic block's start

Parameters:

b - the basic block

Returns:

the child of this interval which starts at or nearest (after) this basic block's start (fist lir instruction index).

- **startsAtBlock**

```
publicintstartsAtBlock()
```

Returns the basic block in which this interval's start position falls.

Returns:

basic block in which this interval's start position falls.

- **endsAtBlock**

```
publicintendsAtBlock()
```

The basic block in which this interval's end position falls.

Returns:

the basic block number.

- **spill**

```
publicvoidspill()
```

Assigns an offset to this interval (if one hasn't been already assigned). Assigns that same offset to any (newly created) children.

- **isChild**

```
publicbooleanisChild()
```

Is this interval a child interval?

Returns:

true or false.

- **isParent**

```
publicbooleanisParent()
```

Is this interval a parent interval? (Ie, does it have children?)

Returns:

true or false.