NLinearRegisterAllocator.java

```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import java.util.ArrayList;

/**
 * Implements the Linear Scan register allocation algorithm.
 */

public class NLinearRegisterAllocator extends NRegisterAllocator {
    /**
     * Interval queues for tracking the allocation process.
     */
    private ArrayList<NInterval> unhandled;
    private ArrayList<NInterval> active;
    private ArrayList<NInterval> inactive;

    /**
     * Used to keep track of which intervals get assigned to what physical
     * register. Needed only in allocateBlockedRegFor.
     */
    private ArrayList<ArrayList<NInterval>> regIntervals;
    private int[] freePos, usePos, blockPos;

    /**
     * Construct a linear register allocator for the given control flow graph.
     *
     * @param cfg
     *            the control flow graph instance.
     */

    public NLinearRegisterAllocator(NControlFlowGraph cfg) {
        super(cfg);
        unhandled = new ArrayList<NInterval>();
        active = new ArrayList<NInterval>();
        inactive = new ArrayList<NInterval>();

        // Instantiate usePositions and freePos to be the size of
        // the physical registers used.
        freePos = new int[NPhysicalRegister.MAX_COUNT];
        usePos = new int[NPhysicalRegister.MAX_COUNT];
        blockPos = new int[NPhysicalRegister.MAX_COUNT];
        regIntervals = new ArrayList<ArrayList<NInterval>>();
        for (int i = 0; i < NPhysicalRegister.MAX_COUNT; i++) {
            regIntervals.add(new ArrayList<NInterval>());
        }
    }

    /**
     * Perform the linear register allocation, assigning physical registers to
     * virtual registers.
     */

    public void allocation() {
        // Build the intervals for the control flow graph.
        this.buildIntervals(); // The correct intervals are now in intervals

        // Add all intervals corresponding to vregs to unhandled list
        for (int i = 32; i < cfg.intervals.size(); i++) {
            this.addSortedToUnhandled(cfg.intervals.get(i));
        }

        // Allocate any fixed registers (a0, ..., a3 and v0) that were
        // assigned during generation phase to the appropriate
        // interval.
```

```
            for (int i = 0; i < 32; i++) {
                if (cfg.registers.get(i) != null) {
                    cfg.intervals.get(i).pRegister = (NPhysicalRegister)
cfg.registers
                            .get(i);
                }
            }

            // Assign stack offset (relative to fp) for formal parameters
            // fourth and above, and stack offset (relative to sp) for
            // arguments fourth or above.
            for (NBasicBlock block : cfg.basicBlocks) {
                for (NLIRInstruction lir : block.lir) {
                    if (lir instanceof NLIRLoadLocal) {
                        NLIRLoadLocal loadLocal = (NLIRLoadLocal) lir;
                        if (loadLocal.local >= 4) {
                            NInterval interval = cfg.intervals
                                    .get(((NVirtualRegister) loadLocal.write)
                                            .number());
                            interval.spill = true;
                            interval.offset = loadLocal.local - 3;
                            interval.offsetFrom = OffsetFrom.FP;
                        }
                    }
                }
            }

            NInterval currInterval; // the current interval
            int psi; // the current interval's first start position
            ArrayList<NInterval> tmp;

            // Linear allocation begins; repeat so long as there are
            // additional virtual registers to map to physical registers.
            while (!unhandled.isEmpty()) {
                currInterval = unhandled.remove(0);
                psi = currInterval.firstRangeStart();
                tmp = new ArrayList<NInterval>();
                for (int i = 0; i < active.size(); i++) {
                    if (active.get(i).lastNRangeStop() < psi) {
                        tmp.add(active.get(i));
                    } else if (!active.get(i).isLiveAt(psi)) {
                        inactive.add(active.get(i));
                        tmp.add(active.get(i));
                    }
                }
                for (NInterval nonActive : tmp) {
                    active.remove(nonActive);
                }
                tmp = new ArrayList<NInterval>();
                for (int i = 0; i < inactive.size(); i++) {
                    if (inactive.get(i).lastNRangeStop() < psi) {
                        tmp.add(inactive.get(i));
                    } else if (inactive.get(i).isLiveAt(psi)) {
                        active.add(inactive.get(i));
                        tmp.add(inactive.get(i));
                    }
                }
                for (NInterval nonInActive : tmp) {
                    inactive.remove(nonInActive);
                }
                if (!this.foundFreeRegFor(currInterval)) {// check
                    this.allocateBlockedRegFor(currInterval); // never fails
                }
                active.add(currInterval);
            }
            this.resolveDataFlow();
        }

        /**
```

```
     * Adds a given interval onto the unhandled list, maintaining an order based
     * on the first range start of the NIntervals.
     *
     * @param newInterval
     *            the NInterval to sort onto unhandled.
     */

    private void addSortedToUnhandled(NInterval newInterval) {
        if (unhandled.isEmpty()) {
            unhandled.add(newInterval);
        } else {
            int i = 0;
            while (i < unhandled.size()
                    && unhandled.get(i).firstRangeStart() <= newInterval
                            .firstRangeStart()) {
                i++;
            }
            unhandled.add(i, newInterval);
        }
    }

    /**
     * Allocates a free physical register for the current interval. Inspects
     * active and inactive sets. Cannot split or alter the assigned physical
     * register of any other interval but current.
     *
     * @param currInterval
     *            the current interval for which a physical register is sought.
     * @return true if a free physical register was found and allocated for
     *         currInterval, false otherwise.
     */

    private boolean foundFreeRegFor(NInterval currInterval) {
        this.initFreePositions(); // must be reset every iteration
        for (NInterval activeInterval : active) {
            if (activeInterval.pRegister != null)
                freePos[activeInterval.pRegister.number - NPhysicalRegister.T0] =
0;
        }
        for (NInterval inactiveInterval : inactive) {
            if (inactiveInterval.nextIntersection(currInterval) >= 0)
                freePos[inactiveInterval.pRegister.number
                        - NPhysicalRegister.T0] = Math.min(
                        freePos[inactiveInterval.pRegister.number
                                - NPhysicalRegister.T0], inactiveInterval
                                .nextIntersection(currInterval));
        }

        // The physical registers available are in NPhysicalRegister.getInfo
        // static array. This is indexed from 0 to NPhysicalRegister.MAX_COUNT
        int reg = this.getBestFreeReg();
        if (freePos[reg] == 0)
            return false;
        else if (freePos[reg] > currInterval.lastNRangeStop()) {
            currInterval.pRegister = NPhysicalRegister.regInfo[reg
                    + NPhysicalRegister.T0];
            cfg.pRegisters.add(NPhysicalRegister.regInfo[reg
                    + NPhysicalRegister.T0]);
            regIntervals.get(reg).add(currInterval);
            return true;
        } else {
            this.addSortedToUnhandled(currInterval.splitAt(freePos[reg]));
            currInterval.spill();
            currInterval.pRegister = NPhysicalRegister.regInfo[reg
                    + NPhysicalRegister.T0];
            regIntervals.get(reg).add(currInterval);
            return true;
        }
    }

    /**
     * Sets all free positions of pregisters available for allocation to a
     * really high number.
     */

    private void initFreePositions() {
        for (int i = 0; i < NPhysicalRegister.MAX_COUNT; i++) {
            freePos[i] = Integer.MAX_VALUE;
        }
    }

    /**
     * The best free physical register.
     *
     * @return the register number.
     */

    private int getBestFreeReg() {
        int freeRegNumber = 0;
        for (int i = 0; i < NPhysicalRegister.MAX_COUNT; i++) {
            if (freePos[i] > freePos[freeRegNumber])
                freeRegNumber = i;
        }
        return freeRegNumber;
    }

    /**
     * Allocates a register based on spilling an interval.
     *
     * @param currInterval
     *            the current interval.
     */

    private void allocateBlockedRegFor(NInterval currInterval) {
        this.initUseAndBlockPositions(); // must be reset every iteration
        for (NInterval activeInterval : active) {
            usePos[activeInterval.pRegister.number - NPhysicalRegister.T0] = Math
                    .min(usePos[activeInterval.pRegister.number
                            - NPhysicalRegister.T0], activeInterval
                            .nextUsageOverlapping(currInterval));
        }
        for (NInterval inactiveInterval : inactive) {
            if (inactiveInterval.nextIntersection(currInterval) >= 0)
                usePos[inactiveInterval.pRegister.number - NPhysicalRegister.T0]
= Math
                        .min(usePos[inactiveInterval.pRegister.number
                                - NPhysicalRegister.T0], inactiveInterval
                                .nextUsageOverlapping(currInterval));
        }
        int reg = this.getBestBlockedReg(); // this is just an index in the
        // usePos array
        if (usePos[reg] < currInterval.firstUsage()) {
            // best to spill current - no reg assignment.
            this.addSortedToUnhandled(currInterval.splitAt(currInterval
                    .firstUsage() - 5));
            currInterval.spill();
            NInterval splitChild = currInterval.splitAt(currInterval
                    .firstRangeStart());
            this.addSortedToUnhandled(splitChild);
            currInterval.spill();
        } else {
            // spilling frees reg for all of current
            currInterval.pRegister = NPhysicalRegister.regInfo[reg
                    + NPhysicalRegister.T0];
            for (NInterval i : regIntervals.get(reg)) {
                if (currInterval.nextIntersection(i) >= 0) {
                    NInterval splitChild = i.splitAt(currInterval
                            .firstRangeStart());
                    this.addSortedToUnhandled(splitChild);
                    i.spill();
                }
            }
            regIntervals.get(reg).add(currInterval);
        }
    }

    /**
     * Initialize use and block positions before processing each virtual
     * rgister.
     */

    private void initUseAndBlockPositions() {
        for (int i = 0; i < NPhysicalRegister.MAX_COUNT; i++) {
            usePos[i] = Integer.MAX_VALUE;
            blockPos[i] = Integer.MAX_VALUE;
        }
    }

    /**
     * Get the best blocked physical register.
     *
     * @return the register number.
     */

    private int getBestBlockedReg() {
        int usableRegNumber = 0;
        for (int i = 0; i < NPhysicalRegister.MAX_COUNT; i++) {
            if (usePos[i] > usePos[usableRegNumber])
                usableRegNumber = i;
        }
        return usableRegNumber;
    }

    /**
     * Resolve the data flow after allocating registers, inserting additional
     * saves and restores for registers to maintain consistency.
     */

    private void resolveDataFlow() {
        // local data flow construction
        // Devised an alternate way of doing this, perhaps with more
        // clarity, will implement later, but has same effect.
        for (NInterval i : cfg.intervals) {
            if (cfg.registers.get(i.vRegId) != null) {
                if (i.spill) {
                    for (int c = 0; c < i.children.size(); c++) {
                        if (i.endsAtBlock() == i.children.get(c)
                                .startsAtBlock()) {
                            if (c == 0) {
                                addStoreInstruction(i, i.lastNRangeStop());
                                addLoadInstruction(i.children.get(c),
                                        i.children.get(c).firstRangeStart());
                            } else {
                                addStoreInstruction(i.children.get(c - 1),
                                        i.children.get(c - 1).lastNRangeStop());
                                addLoadInstruction(i.children.get(c),
                                        i.children.get(c).firstRangeStart());
                            }
                        }
                    }
                }
            }
        }

        // resolution of global data flow
        for (NBasicBlock b : cfg.basicBlocks) {
            for (NBasicBlock s : b.successors) {
                for (int i = s.liveIn.nextSetBit(0); i >= 0; i = s.liveIn
                        .nextSetBit(i + 1)) {
                    NInterval parent = cfg.intervals.get(i);
                    NInterval from = parent.childAtOrEndingBefore(b);
                    NInterval to = parent.childAtOrStartingAfter(s);
                    if (!from.equals(to)) {
                        addStoreInstruction(from, from.usePositions.floorKey(b
                                .getLastLIRInstId()));
                        to = getSegmentWithNearestUse(to, s.getFirstLIRInstId());
                        if (to.usePositions.ceilingEntry(s.getFirstLIRInstId())
                                .getValue() == InstructionType.read)
                            // no use loading prior to a write.
                            addLoadInstruction(to, to.usePositions.ceilingKey(s
                                    .getFirstLIRInstId()));
                    }
                }
            }
        }
    }

    /**
     * Get the the interval segment that contains the nearest first use.
     *
     * @param i
     *             the interval segment (could be a parent or child).
     * @param id
     *             the lir id after which a use is sought.
     * @return the interval segment that that contains the first use at or after
     *          the id position and is associated with the interval i through a
     *          sibling or child relationship. Returns i if there is a use after
     *          id within i. Null if no interval exists that is related to i and
     *          contains a use position at or after id.
     */

    private NInterval getSegmentWithNearestUse(NInterval i, int id) {
        if (i.usePositions.ceilingEntry(id) != null)
            return i;
        else {
            NInterval parent = i;
            int idx = 0;
            if (i.isChild()) {
                parent = i.parent;
                idx = parent.children.indexOf(i) + 1;
            }
            for (; idx < parent.children.size(); idx++) {
                if (parent.children.get(idx).usePositions.ceilingEntry(id) !=
null)
                    return parent.children.get(idx);
            }
            return null;
        }
    }

    /**
     * Adds a store instruction right after a use position specified by id.
     *
     * @param from
     *             the interval which this use position is a part of.
     * @param id
     *             the id of the use position.
     */

    private void addStoreInstruction(NInterval from, int id) {
        NBasicBlock b = cfg.blockAt(id);
        id++;
        if (b.idIsFree(id)) { // assumes always same instr
            b.insertLIRInst(new NLIRStore(b, id, from.offset, from.offsetFrom,
                    from.pRegister));
        }
```

```java
    }

    /**
     * Adds a store instruction right before a use position specified by id.
     *
     * @param to
     *            the interval which this use position is a part of.
     * @param id
     *            the id of the use position.
     */

    private void addLoadInstruction(NInterval to, int id) {
        NBasicBlock s = cfg.blockAt(id);
        id--;
        if (s.idIsFree(id)) { // assumes always same instr
            s.insertLIRInst(new NLIRLoad(s, id, to.offset, to.offsetFrom,
                    to.pRegister));
        }
    }

}
```