Register Allocation in Just-in-Time Compilers: 15 Years of Linear Scan

Kevin Millikin—Google
13 December 2013
Register Allocation Overview
Register allocation

Intermediate representation (IR): arbitrarily many virtual registers

Processor: fixed number (k) of registers

Allocate virtual registers to registers or memory

Registers are faster than memory
Liveness analysis

A value is *live at a program point* if it is read before being written to on some path to an exit.

Data-flow analysis: for each program point what is the set of live values (virtual registers)?

Live values need to be stored somewhere
Interference graph

Two virtual registers *interfere* if they are both live at some program point

Vertexes: all virtual registers
Edges: all interferences

Interferences cannot get the same register
Graph \((k)\)-coloring

Assign colors to vertexes

No two adjacent vertexes can have the same color

If impossible, delete (spill) a vertex and retry

A coloring gives a register assignment
Too slow for just-in-time

Graph coloring is NP-complete for $k > 2$

Plus spilling

This is too slow
Linear Scan Register Allocation

Liveness analysis

for each program point, what is the set of live values?

≈

for each value, what is the set of program points where it is live?

Focus on the virtual registers
Linearize the program

Number the instructions consecutively

Linear scan works for any ordering

But the specific ordering is important

Program points are totally ordered
Approximate liveness analysis

for each value,
what is the set of program points where it is live?

⊆

for each value,
what is the earliest and latest program points where it is live?

This gives a live range per virtual register
Key ideas

Sort the live ranges by start point

Consider them for allocation in sorted order

The number of needed registers only changes when a live range starts or ends

Use a greedy strategy to assign registers
Allocate a free register

class Interval { int start, end, register }

Interval active[k] ← { null, … }

bool AllocateFreeRegister(Interval interval)

for i ← 0 to k-1

    if (active[i] = null) then
        active[i] ← interval

    return true

return false
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
void SpillAtStart(Interval interval)
    spill ← -1, latest ← interval.end
    for i ← 0 to k-1
        if (active[i].end > latest) then
            spill ← i, latest ← active[i].end
    if (spill = -1) then
        interval.register ← -1
    else
        active[spill].register ← -1
        active[spill] ← interval
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
void ExpireOldIntervals(int point)

    for i ← 0 to k-1

        if (active[i].end ≤ point) then
            active[i].register ← i
            active[i] ← null
Example (k=3)
Example (k=3)
Example (k=3)
Putting it all together

```java
void LinearScan(Interval intervals[])
    intervals.sort();
    for i in intervals
        ExpireOldIntervals(i.start)
        if (!AllocateFreeRegister(i)) then
            SpillAtStart(i)
        ExpireOldIntervals(intervals.last().end)
```
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Instruction ordering

Poletto and Sarkar: reverse postorder

Also: code generation order

There was negligible impact
Spill selection

Poletto and Sarkar: latest ending point

Also: weight based on (estimated) use count

There was negligible impact
Lifetime holes

Intervals within a live range that do not contain a useful value

Example: between a load and a store, or a basic block where a virtual register is not used

Linear scan does not consider lifetime holes
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Second-chance allocation

With linear scan a virtual register is spilled for its entire live range

Instead: split and spill after the split, insert a move

Add the spilled live range to the unallocated list

Spilled values get a chance to get a register later
Spilling heuristic

Latest next use (Poletto and Sarkar used latest ending point)

Weighted by loop nesting depth of uses
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Example (k=3)
Spill store elimination

The spilled value could be the same as the current value in a spill slot (e.g., it was previously spilled)

Track each spill slot, do not spill unless needed

Can be inconsistent if a value is not spilled on all paths, use a data-flow analysis and extra stores to guarantee consistency
Resolution

Control flow is not really linear

Different allocation decisions on different paths (e.g., register/spill or register/register)

Resolved at control-flow joins by moves in the predecessor blocks

Implemented by iterating edges after allocation
Example
Example

Blue←Green

A

Blue←Spill

B

C
Example

A

B

C

Blue←Green
Green←Blue

Blue←Spill
Linear Scan in the “Context” of SSA Form

Static single assignment form

Each virtual register is assigned once, extra virtual registers are used.

Phi functions are inserted at control flow joins to merge multiple incoming values.

Translation out of SSA is performed before register allocation, by inserting moves in phi predecessor blocks.
Liveness analysis

Same as linear scan, except consider lifetime holes

Live ranges are explicitly represented as a collection of intervals
Coalesce virtual registers

Before allocation, live ranges are joined if they should share a register

Example: phi moves

Example: x86 two-operand instructions (e.g. ADD)
Inactive intervals

In addition to active intervals (allocated to a register and live)

Inactive intervals are ones allocated to a register but in a lifetime hole

Inactive intervals are tracked separately
Expiring old intervals

For each active interval:

If ended, the interval is expired

If a lifetime hole is reached, the interval is moved to inactive
Reactivate inactive intervals

For each inactive interval:

If ended, the interval is expired

If the end of a lifetime hole is reached, the interval is moved to active
Allocate a free register

Same as linear scan, except

A register is not free at a position if it will contain a live part of an inactive interval
Spill an interval

Same as linear scan, except

Inactive intervals are also spill candidates
Resolution

No splitting

So no resolution is required
Allocate a free register

Same as ‘in the context of SSA’, except

Registers containing inactive intervals are free, but

The allocated interval must be split at the point that the lifetime hole ends
Resolution

Splitting

So, resolution is required again
Optimal split positions

Remember: second-chance binpacking split as late as possible

Instead: move splits out of loops

Also: move splits to basic block boundaries
Register hints

“Lightweight” coalescing

Record the register of the first range as a hint to the second

Hints are preferred but can be ignored
Linear Scan on SSA Form

Resolution

Recall: phi moves inserted in join predecessor blocks to translate out of SSA

Recall: resolution due to splitting inserts moves in join predecessor blocks

Idea: do allocation on SSA form \textit{then} translate out, needing only one set of moves
Conclusion
Linear scan

State of the art for JIT compilers

Mature, though relatively new

Easy to implement in a simple form

Modern implementations are quite sophisticated

Performance approaches graph coloring