Linear Scan Register Allocation

on Static Single Assignment Form

Christian Wimmer
cwimmer@uci.edu
www.christianwimmer.at
July 2010

Department of Computer Science
University of California, Irvine
Background

Department of Computer Science
University of California, Irvine

- SSA Based Register Allocation
- Trace Based Compilation
- Phase Change Detection
- Hierarchical Layering of VMs
- Information Flow for JavaScript
- Multivariant Execution

Institute for System Software
Johannes Kepler University Linz, Austria

- Linear Scan Register Allocation
- Automatic Object Inlining
- Array Bounds Check Elimination
- Optimization of Strings
- Continuations and Coroutines
- Dynamic Code Evolution

Feedback-Directed Optimistic Optimizations in Virtual Machines
Why Still (or Again) Register Allocation?

Client
Methods: 6788
Bytecodes: 1120 KB
Code Size: 4690 KB
Speed: 265 KB/Sec.

Server
Methods: 4674
Bytecodes: 1985 KB
Code Size: 7638 KB
Speed: 24 KB/Sec.

SPECjvm2008, Lagom, all benchmarks w/o SciMark
2 * Intel Xeon X5140, 2.33 GHz, 4 cores, 32 GByte memory
Register Allocation and SSA Form

- Register allocation
  - Graph coloring algorithm
  - Linear scan algorithm

- Static single assignment (SSA) form
  - One definition per variable that dominates all uses
    - Variable alive continuously from this single definition to all uses
    - Dead variables never become alive again spuriously
    - The “corner case” examples of previous papers are impossible
  - Interference graph is chordal
    - Graph coloring in polynomial time
  - Variables that interfere somewhere also interfere at one definition
    - Enough to check interference once at definition point
    - No explicit interference graph necessary
Algorithm 4.2 Coloring an interference graph of a SSA-form program

procedure Color-Program(Program P)
    Color-Recursive(start block of P)

procedure Color-Recursive(Basic block $B = \langle \ell_1, \ldots, \ell_n \rangle$)
    for all $x \in livein(B)$ do
        assigned ← assigned $\cup \rho(x)$  // All variables live in have already been colored
    for $i \leftarrow 1$ to $n$ do
        for all $x \in arg(\ell_i)$ do
            if last use of $x$ then
                assigned ← assigned $\setminus \rho(x)$
        for all $y \in res(\ell_i)$ do
            $\rho(y) \leftarrow$ one of $R \setminus assigned$
    for \{$C \mid B = idom(C)$\} do
        Color-Recursive($C$)  // Proceed with all children in the dominance tree

[Hack 2007, PhD Thesis]
Java HotSpot™ VM

Execution Engine

Class Loader \(\rightarrow\) Bytecodes

Interpreter

JIT Compiler\(\rightarrow\) Machine Code

executes

compiles

generates

access memory

Heap

Young Generation

Old Generation

Eden \(\rightarrow\) From \(\rightarrow\) To

Old Space

Garbage Collector

Stop-and-Copy

Mark-and-Compact

collects

collects
Java HotSpot™ Client Compiler

Bytecodes

Method Inlining
Constant Propagation
Local Value Numbering

HIR (SSA Form)

Null Check Elimination
Global Value Numbering

Optimized HIR

Global Optimizations

Bytecode Parsing

LIR Generation

LIR

Register Allocation

Code Generation

Machine Code

Back End
Phases of Linear Scan Algorithm

**Linear Scan not on SSA Form**
- SSA Form Deconstruction
  - LIR not in SSA Form
- Requires a Data Flow Analysis
  - Lifetime Intervals
- Splitting and Spilling of Intervals
  - Registers Assigned to Intervals
- Linear Scan Algorithm
  - LIR not in SSA Form
- Resolution
  - LIR not in SSA Form

**Linear Scan on SSA Form**
- LIR Generation
  - LIR in SSA Form
- Lifetime Analysis
  - Lifetime Intervals
- Splitting and Spilling of Intervals
  - Registers Assigned to Intervals
- Linear Scan Algorithm
  - Splitting and Spilling of Intervals
- Resolution
  - SSA Form Deconstruction
**Lifetime Intervals Without SSA Form**

![Graphical representation of lifetime intervals without SSA form]

Legend:
- **Lifetime Interval**
- **Definition position**
- **Use position**

- **20: move 1 -> R12**
- **22: move R11 -> R13**
- **24: label B2**
- **26: cmp R13, 1**
- **28: branch lessThan B4**
- **30: label B3**
- **32: mul R12, R13 -> R14**
- **34: sub R13, 1 -> R15**
- **36: move R14 -> R12**
- **38: move R15 -> R13**
- **40: jump B2**
- **42: label B4**

*define R10, R11*

*use R10, R12*
Lifetime Intervals With SSA Form

define R10, R11

20: label B2
   phi [1, R14] -> R12
   phi [R11, R15] -> R13
22: cmp R13, 1
24: branch lessThan B4

26: label B3
28: mul R12, R13 -> R14
30: sub R13, 1 -> R15
32: jump B2

34: label B4
   use R10, R12

---

Lifetime Intervals

- **Lifetime Interval**
  - Light gray rectangles
- **Definition position**
  - Red rectangles
- **Use position**
  - Green rectangles
Construction of Lifetime Intervals

- Initial Live Set from Successors
- Add Input Operands of Successors’ Phis
- Process Operations in Reverse Order
- Remove Phi Functions from Live Set
- Extend Live Ranges of Loop Variables

```
20: label B2
   phi [1, R14] -> R12
   phi [R11, R15] -> R13
22: cmp R13, 1
24: branch lessThan B4

26: label B3
28: mul R12, R13 -> R14
30: sub R13, 1 -> R15
32: jump B2

34: label B4
   use R10, R12
```

- define R10, R11
Irreducible Control Flow

phi \([R10, R12] \rightarrow R11\)
phi \([R10, R11] \rightarrow R12\)
phi \([R20, R22] \rightarrow R21\)
phi \([R20, R21] \rightarrow R22\)
Linear Scan Algorithm

**LinearScan**

`unhandled = list of intervals sorted by increasing start positions`
`active = {}; inactive = {}; handled = {}`

while `unhandled ≠ {}` do
  `current = pick and remove first interval from unhandled`
  `position = start position of current`

  // check for intervals in active that are handled or inactive
  for each interval `it in active` do
    if `it ends before position` then
      move it from active to handled
    else if `it does not cover position` then
      move it from active to inactive

  // check for intervals in inactive that are handled or active
  for each interval `it in inactive` do
    if `it ends before position` then
      move it from inactive to handled
    else if `it covers position` then
      move it from inactive to active

  // find a register for current
  TRY_ALLOCATE_FREE_REG
  if allocation failed then ALLOCATE_BLOCKED_REG

  if `current has a register assigned` then add `current` to active

**TRY_ALLOCATE_FREE_REG**

set `freeUntilPos` of all physical registers to `maxInt`

for each interval `it in active` do
  `freeUntilPos[it.reg] = 0`

for each interval `it in inactive` intersecting with `current` do
  if `freeUntilPos[it.reg] = next intersection of it with current`

  `reg = register with highest freeUntilPos`
  if `freeUntilPos[reg] = 0` then
    // no register available without spilling
    allocation failed
  else if `current ends before freeUntilPos[reg]` then
    // register available for the whole interval
    `current.reg = reg`
  else
    // register available for the first part of the interval
    `current.reg = reg`
    split `current` before `freeUntilPos[reg]`
Changes to Linear Scan Algorithm

Linear scan not on SSA form

- $i_{10}$ has lifetime hole
- $i_{10}$ and $i_{11}$ can intersect

Without SSA form:
Intervals that are currently not live can block registers

Linear scan on SSA form

- $i_{10}$ and $i_{11}$ never intersect
- $i_{10}$ has lifetime hole

SSA form guarantees:
Intervals that are currently not live never block registers
SSA Deconstruction during Resolution

Resolution
Visit intervals live across control-flow edges

SSA Deconstruction
Also visit intervals starting at the control-flow edge

B2 – B4:
No move necessary

B3 – B4:
move s1 -> eax
move s2 -> ecx
Compilation Time

Compilation time of baseline and SSA form version of linear scan

Resolution
Linear Scan
Lifetime Analysis
LIR Construction

SPECjvm2008
SPECjbb2005
DaCapo
SciMark

2 * Intel Xeon X5140, 2.33 GHz, 4 cores, 32 GByte memory
Ubuntu Linux, kernel version 2.6.28
SPECjvm2008: Lagom w/o SciMark
## Phi Functions and Move Instructions

<table>
<thead>
<tr>
<th></th>
<th>DaCapo Baseline</th>
<th>DaCapo SSA Form</th>
<th>SciMark Baseline</th>
<th>SciMark SSA Form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before Register Allocation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moves</td>
<td>402,678</td>
<td>355,936</td>
<td>-12%</td>
<td>908</td>
</tr>
<tr>
<td>Phi Functions</td>
<td>0</td>
<td>20,542</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td><strong>After Register Allocation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moves Register to Register</td>
<td>127,318</td>
<td>124,351</td>
<td>-2%</td>
<td>193</td>
</tr>
<tr>
<td>Moves Constant to Register</td>
<td>71,967</td>
<td>70,663</td>
<td>-2%</td>
<td>99</td>
</tr>
<tr>
<td>Moves Stack to Register</td>
<td>3,718</td>
<td>3,722</td>
<td>+0%</td>
<td>12</td>
</tr>
<tr>
<td>Moves Register to Stack</td>
<td>65,973</td>
<td>56,639</td>
<td>-14%</td>
<td>166</td>
</tr>
<tr>
<td>Moves Constant to Stack</td>
<td>0</td>
<td>1,386</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Moves Stack to Stack</td>
<td>0</td>
<td>647</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Future Work

**Traditional Graph Coloring**
- Slow, iterative algorithm
- Good code quality

**Graph Coloring on SSA Form**
- Interference graph not necessary for allocation, but used for global spilling decisions

**Hybrid Solution**
- Take the best from both worlds:
  - Global spilling decisions using interference graph.
  - Some local decisions for spilling and fixed registers.
  - Fast and good code quality?

**Traditional Linear Scan**
- Fast, linear algorithm
- Good enough code quality

**Linear Scan on SSA Form**
- Lifetime intervals not necessary for allocation, but used for spilling decisions and fixed registers