Linear Scan Register Allocation

by Massimiliano Poletto and Vivek Sarkar
presented by Ivaylo Dobrikov

January 8, 2013
The linear scan algorithm is a simple algorithm for global register allocation.

The linear scan algorithm is efficient and produces relatively good code.
Program Model I

- Intermediate representation:
  - RTL-like quads
  - Arithmetic operations are performed directly on the virtual registers
  - No load/store instructions for accessing virtual registers
- No live range splitting
- Pseudo-instructions are numbered according to some order
Program Model (Instruction Orderings)

Instructions Ordering

if A then
    B
else
    C
D
Program Model (Instruction Orderings)

Instructions Ordering

if A then
  B
else
  C
D

- order in which the pseudo-instructions appear in the IR:
**Program Model (Instruction Orderings)**

Instructions Ordering

```plaintext
if A then
  B
else
  C
D
```

- breadth-first order:

```
(1) A
(2) B
(3) C
(4) D
```
Program Model (Instruction Orderings)

Instructions Ordering

if A then
  B
else
  C
D

- depth-first order:
Program Model (Instruction Orderings)

Note: The order does not affect the correctness of the algorithm, but it may affect the quality of allocation.
Program Model (Live Intervals)

For some given numbering of the IR

![Diagram: live interval from i to j]

is the live interval of \( v \) if there is no \( j' > j \) such that \( v \) is live at \( j' \) and if there is no \( i' < i \) such that \( v \) is live at \( i' \).

The trivial live interval for any variable is \([1,N]\). (\( N \) is the number of the instructions in the IR)
Data-flow-graph Example (Live Intervals)

\[
\begin{align*}
    b &= a + 1 \\
    c &= a + b \\
    d &= b + c \\
    a &= a + 1 \\
    e &= b - d \\
    b &= d + e \\
    c &= d - 1 \\
    b &= c + 1
\end{align*}
\]
Data-flow-graph Example (Live Intervals)

\[
\begin{align*}
(1) & \quad b = a + 1 \\
(2) & \quad c = a + b \\
(3) & \quad d = b + c \\
(4) & \quad a = a + 1 \\
(5) & \quad e = b - d \\
(6) & \quad b = d + e \\
(7) & \quad c = d - 1 \\
(8) & \quad b = c + 1
\end{align*}
\]
Data-flow-graph Example (Live Intervals)

(1) \( b = a + 1 \)  
(2) \( c = a + b \)  
(3) \( d = b + c \)  
(4) \( a = a + 1 \)  
(5) \( e = b - d \)  
(6) \( b = d + e \)  
(7) \( c = d - 1 \)  
(8) \( b = c + 1 \)
Data-flow-graph Example (Live Intervals)

(1) b = a + 1
(2) c = a + b
(3) d = b + c
(4) a = a + 1
(5) e = b - d
(6) b = d + e
(7) c = d - 1
(8) b = c + 1

a: [1, 4]
Data-flow-graph Example (Live Intervals)

\[
\begin{align*}
(1) & \quad b = a + 1 \\
(2) & \quad c = a + b \\
(3) & \quad d = b + c \\
(4) & \quad a = a + 1 \\
(5) & \quad e = b - d \\
(6) & \quad b = d + e \\
(7) & \quad c = d - 1 \\
(8) & \quad b = c + 1
\end{align*}
\]

Live Intervals:
- \(a\): [1,4]
- \(b\): [2,5]
Data-flow-graph Example (Live Intervals)

1. $b = a + 1$
2. $c = a + b$
3. $d = b + c$
4. $a = a + 1$
5. $e = b - d$
6. $b = d + e$
7. $c = d - 1$
8. $b = c + 1$

Live intervals:
- $a$: [1, 4]
- $b$: [2, 5]
- $c$: [3, 8]
Data-flow-graph Example (Live Intervals)

1. \( b = a + 1 \)
2. \( c = a + b \)
3. \( d = b + c \)
4. \( a = a + 1 \)
5. \( e = b - d \)
6. \( b = d + e \)
7. \( c = d - 1 \)
8. \( b = c + 1 \)

- \( a \): [1,4]
- \( b \): [2,5]
- \( c \): [3,8]
- \( d \): [5,7]
Data-flow-graph Example (Live Intervals)

1. \( b = a + 1 \)
2. \( c = a + b \)
3. \( d = b + c \)
4. \( a = a + 1 \)
5. \( e = b - d \)
6. \( b = d + e \)
7. \( c = d - 1 \)
8. \( b = c + 1 \)

Live Intervals:
- \( a: [1,4] \)
- \( b: [2,5] \)
- \( c: [3,8] \)
- \( d: [5,7] \)
- \( e: [6,7] \)
Linear Scan Register Allocation

Compilation Phase

- Live Variable Analysis
- Live Interval Construction
- Linear Scan Register Allocation
The Linear Scan Algorithm

See Whiteboard
Example: Setting Live Intervals

\begin{center}
\begin{tikzpicture}
\draw (0,0) node{a} -- (1,0) node{b} -- (2,0) node{c} -- (3,0) node{d} -- (4,0) node{e};
\draw (0,-1) node{1} -- (1,-1) node{2} -- (2,-1) node{3} -- (3,-1) node{4} -- (4,-1) node{5} -- (5,-1) node{6} -- (6,-1) node{7} -- (7,-1) node{8};
\draw (6,0) node{active=<>};
\end{tikzpicture}
\end{center}
Example: Setting Live Intervals
Example: Setting Live Intervals

The Linear Scan Algorithm

Algorithm

Example: Setting Live Intervals
Example: Setting Live Intervals

active = \langle a, b \rangle
Example: Setting Live Intervals

active = \langle a, b \rangle
Example: Setting Live Intervals

active=⟨a,b⟩, endpoint[b]>endpoint[c]?
Example: Setting Live Intervals

active=⟨a,b⟩, spill c
Example: Setting Live Intervals

active = ⟨a, b⟩
Example: Setting Live Intervals

```
active=<b>
```

Diagram showing intervals for elements a, b, c, d, e over the sequence 1, 2, 3, 4, 5, 6, 7, 8.
Example: Setting Live Intervals

active = <b, d>
Example: Setting Live Intervals

active = <b, d>
Example: Setting Live Intervals

active=⟨d⟩
Example: Setting Live Intervals

```
Example: Setting Live Intervals

active=<e,d>
```

Diagram:

```
[Diagram showing live intervals for characters a to e, with active interval indicated]
```
Example: Setting Live Intervals

\[ \text{active} = \langle e, d \rangle \]
Complexity

**LineaScanRegisterAllocation**

```
active ← {}
foreach live interval \( i \), in order of increasing start points
    ExpireOldInterval(i)
    if length(active) = R then
        SpillAtInterval(i)
    else
        register[i] ← a register removed from pool of free registers
        add \( i \) to active, sorted by increasing endpoint
```

- \( O(V) \)
  - \( V \): number of variable candidates
  - \( R \): is assumed to be constant
- \( O(V \times \log(R)) \) or \( O(V \times R) \) if \( R \) is not constant
Methodology (ICODE Infrastructure)

The ICODE Infrastructure:

- runtime system of the **tcc** dynamic compiler
- **tcc** is an extension of ANSI C that enables dynamic code generation
- produces good quality code with low compile-time overhead
- used primarily for Compile-Time Performance measurements
- C’ benchmarks for the ICODE Implementation are relative small, therefore the run-time performance is similar for the different register allocation approaches (These benchmark evaluations will not be discussed in this presentation!)
Methodology (SUIF Infrastructure)

The SUIF Infrastructure:

- used primarily for Run-Time Performance measurements
- SUIF implementation used to compile various SPEC benchmarks
Global Register Allocation Approaches

- Abstracting the register allocation problem as a graph coloring problem
  - eliminating redundant moves via *coalescing*
- Register allocation with the highest estimated usage counts
- Second-chance bin-packing register allocation
Usage Counts Algorithm

- Allocation of registers:
  - allocating registers to the variables with the highest estimated usage counts
  - all other variables will be placed on the stack
- Important for the Performance Evaluations: No Live variable analysis and no Allocation setup used for allocating registers to variables.
Graph Coloring Approach

- **Idea:** Abstracting the register allocation problem as a graph coloring problem.
- **Nodes in the interference graph** represent live ranges.
- **Nodes connected by an edge** when live ranges simultaneously live at at least one program point.
- **Any two connected nodes** receive two different colors.
- **If graph not colorable nodes must be spilled.**
- **Basic goal:** Finding legal coloring after deleting the minimum number of nodes.
Graph Coloring Approach Example

R = 2, V = 4:

Code Sequence
...
a=3
b=a*8
c=a+b
d=a+c
a=c*d
a=c*d
...
Graph Coloring Approach Example

\( R = 2, V = 4: \)

**Code Sequence**

...  
\( a = 3 \)  
\( b = a \times 8 \)  
\( c = a + b \)  
\( d = a + c \)  
\( a = c \times d \)  
...
Graph Coloring Approach (Optimizations)

- Using the coalescing technique to eliminate redundant moves:
  - When source and destination of a move instruction do not share an edge, then the coalescing nodes can be coalesced into one, and move can be eliminated.

- Problem: aggressive coalescing can lead to non-colorable graphs.
Graph Coloring Register Allocation

Compilation Phase

Live Variable Analysis

Register Coalescing

Graph Coloring
Recall Linear Scan Register Allocation

Compilation Phase

Live Variable Analysis

Live Interval Construction

Linear Scan Register Allocation
Second-Chance Bin-Packing

It will be discussed later!
ICODE Implementation (Compiler-Time Performance)

Fig. 3. Register allocation overhead for dynamic code ('C') kernels. U denotes a simple algorithm based on usage counts. L denotes linear scan. C denotes graph coloring.
## SUIF Implementation (Compiler-Time Performance)

<table>
<thead>
<tr>
<th>File (Benchmark)</th>
<th>Linear scan</th>
<th>Bin-Packing</th>
<th>Ratio (Binp./lin. scan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim.f (swim)</td>
<td>0.42s</td>
<td>1.07s</td>
<td>2.55</td>
</tr>
<tr>
<td>xlist.c (li)</td>
<td>0.31s</td>
<td>0.60s</td>
<td>1.94</td>
</tr>
<tr>
<td>xleval (li)</td>
<td>0.14s</td>
<td>0.29s</td>
<td>2.07</td>
</tr>
<tr>
<td>tomcatv.f (tomcatv)</td>
<td>0.19s</td>
<td>0.48s</td>
<td>2.53</td>
</tr>
<tr>
<td>compress.c (compress)</td>
<td>0.14s</td>
<td>0.32s</td>
<td>2.29</td>
</tr>
<tr>
<td>cvrin.c (espresso)</td>
<td>0.61s</td>
<td>1.14s</td>
<td>1.87</td>
</tr>
<tr>
<td>backprop.c (alvinn)</td>
<td>0.07s</td>
<td>0.19s</td>
<td>2.71</td>
</tr>
<tr>
<td>fpppp.f (fpppp)</td>
<td>3.35s</td>
<td>4.26s</td>
<td>1.27</td>
</tr>
<tr>
<td>twldrv.f (fpppp)</td>
<td>1.70s</td>
<td>3.49s</td>
<td>2.05</td>
</tr>
</tbody>
</table>
Pathological Case 1 (Compile-Time Performance)
Pathological Case 1 (Compile-Time Performance)

Fig. 5. Overhead of graph coloring and linear scan as a function of the number of simultaneously live variables for programs of type (a).
Pathological Case 2 (Compile-Time Performance)
Pathological Case 2 (Compile-Time Performance)

Fig. 6. Overhead of graph coloring and linear scan as a function of program size for programs of type (b). The horizontal axis denotes the number of staggered sets of intervals ($k$ in Figure 4(b)). Different curves denote values for different numbers of simultaneously live variables ($m$ in Figure 4(b)).
Second-Chance Bin-Packing

- More complex linear scan algorithm
- Evolution and refinement of bin-packing (objects of different volumes must be packed into a finite number of bins of capacity $V$ in a way that minimizes the number of bins used)
- Invest more time in compilation in order to generate better code.
- Variable’s lifetime can be split multiple times (variable resides in a register in some parts of the program and in memory in other parts).
- The Second-Chance Bin-Packing Algorithm does more work at compile time, but can emit better code than the Linear Scan Algorithm.
Data-flow-graph Example (Multiple Live Intervals)

```
(1) b = a + 1
(2) c = a + b
(3) d = b + c
(4) a = a + 1
(5) e = b - d
(6) b = d + e
(7) c = d - 1
(8) b = c + 1
```

c: [3,3] and c: [8,8]
Fig. 8. Run times of static C benchmarks compiled with different register allocation algorithms. U, L, and C are as before. B denotes second-chance binpacking.
What do we observed from the Performance Benchmarks?

- The linear scan algorithm is significantly faster than graph coloring algorithms.
- The linear scan algorithm generally emits code that runs within approximately 10% of the speed of that generated by an aggressive graph coloring algorithm.
Fast Live Interval Analysis

- SCC-based liveness analysis
- Advantages: more quick compiling than the full live variable analysis
- Disadvantages: bad quality of the generated code for large programs
 Compile-Time Performance (SCC-based vs. full liveness analysis)

**Fig. 10.** Comparison of register allocation overhead of linear scan with full live variable analysis (L) and SCC-based liveness analysis (S).
## Run-Time Performance (SCC-based vs. full liveness analysis)

<table>
<thead>
<tr>
<th>File (Benchmark)</th>
<th>SCC-based analysis</th>
<th>Full liveness analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>22.7s (6.68)</td>
<td>4.0s (1.18)</td>
</tr>
<tr>
<td>compress</td>
<td>134.4s (3.49)</td>
<td>43.1s (1.12)</td>
</tr>
<tr>
<td>li</td>
<td>14.2s (2.90)</td>
<td>5.4s (1.10)</td>
</tr>
<tr>
<td>alvinn</td>
<td>40.2s (1.73)</td>
<td>24.8s (1.06)</td>
</tr>
<tr>
<td>tomcatv</td>
<td>290.8s (5.09)</td>
<td>60.5s (1.06)</td>
</tr>
<tr>
<td>swim</td>
<td>303.5s (7.38)</td>
<td>44.6s (1.09)</td>
</tr>
<tr>
<td>fpppp</td>
<td>484.7s (5.43)</td>
<td>90.8s (1.02)</td>
</tr>
<tr>
<td>ec</td>
<td>23.2s (5.80)</td>
<td>5.7s (1.43)</td>
</tr>
<tr>
<td>sort</td>
<td>10.6s (3.21)</td>
<td>3.5s (1.06)</td>
</tr>
</tbody>
</table>
Spilling Heuristics

- Spilling heuristic based on interval weight (estimated usage count).
  - Spilling the interval with the least estimated usage count.
- Disadvantages: maintaining usage count information
- Compile-Time Performances similar to the interval length spilling approach
Optimizations

- Live interval splitting: Instead of one interval for the entire flow graph, each variable has one live interval for each region of the flow graph throughout which it is uninterruptedly live.
- Advantages: better quality of the emitted code.
- Disadvantages: slower compile-time.
Poletto, M.; Sarkar, V.:
Linear Scan Register Allocation
1998.

Aho, A. V.; Lam, M.S.; Sethi, R.; Ullman, J.D.:
Compiler: Prinzipien, Techniken und Werkzeuge