Register Allocation Deconstructed

David Ryan Koes
Seth Copen Goldstein

Carnegie Mellon

12th International Workshop on Software and Compilers for Embedded Systems
April 24, 2009
Register Allocation Problem

unbounded number of program variables
...
\[ v = 1 \]
\[ w = v + 3 \]
\[ x = w + v \]
\[ u = v \]
\[ t = u + x \]
print(x);
print(w);
print(t);
print(u);
...

limited number of processor registers + slow memory

register allocator

eax
ebx
ebx
ecx
edx
esi
edi
esp
ebp
Register Allocation

- Graph coloring
- Linear scan
- Optimal frameworks
- “Move elimination” allocators

- Spill Code Optimization
  - spill to memory to meet register availability

- Move Insertion
  - insert moves to make assignment easy (or leave in SSA form)

- Assignment
  - assign variables to registers; attempt to maximize move coalescing
Questions

- What is the penalty of decomposing register allocation into individual components?
- What is the individual impact of each component on code quality?
- How far from optimal are existing heuristics?

Our goal is to answer these questions.

An optimal register allocation framework is used to empirically evaluate the importance of the components of register allocation, the impact of component integration, and the effectiveness of existing heuristics.
Outline

• Motivation
• Register Allocation Components
  – Move Insertion
  – Coalescing
  – Spilling
  – Assignment
• Methodology
• Results
• Conclusion
Move Insertion

• Additional move instructions can simplify assignment problem
• Can eliminate need to spill
• Only indirect impact on code quality

L1:
write b
read a
write c
read b
write a
read c
branch L1

move r0 -> r1
Move Insertion Evaluation

**full:** move instructions may be inserted at any program point

**limited:** move instructions may be inserted only at the entry and exit of basic blocks

**none:** no register-to-register move instructions are generated by the allocator
Coalescing

- Eliminate move instructions by assigning each operand to the same location
- Can be performed as separate pass
  - lose ability to coalesce with physical registers
  - lose ability to coalesce “uncoalescables”
Coalescing Evaluation

**integrated optimal**: move coalescing is solved optimally as part of the complete register allocation problem

**integrated optimal ignoring uncoalescable**: the register allocator fully optimizes only those move instructions identified as coalescable prior to register allocation

**separate optimal**: move coalescing is solved optimally as a separate problem prior to allocation

**separate aggressive**: a greedy heuristic aggressively eliminates coalescable moves prior to register allocation

**none**: no coalescing is performed
Spilling

• Can be performed as a separate pass
  – spill variables to memory to meet register needs at each program point
  – if move and swap insertions are allowed, assignment is now possible

MAXLIVE ≤ #REG = 2
Spilling Evaluation

**integrated optimal:** spill code generation is solved optimally as part of the complete register allocation problem

**separate optimal:** the spill code generation problem (reducing max liveness to meet register availability) is solved optimally as a standalone problem

**separate heuristic:** the spill code generation problem is solved as a standalone problem using a heuristic algorithm
Assignment

• assign physical register(s) to each variable at every program point
• may change assignment of variable by inserting move instruction
• if spilling and coalescing are performed separately, leaves assignment
• optimizes for register preferences
Assignment Evaluation

**integrated optimal:** assignment is solved optimally as part of the complete register allocation problem

**graph heuristic:** a graph-coloring based heuristic is used to assign registers to the results of spill code generation; move instructions may be inserted to improve colorability

**linear scan heuristic:** a linear scan based heuristic is used to assign register to the results of spill code generation; move instructions may be inserted to improve colorability
Methodology

Implement optimal register allocation framework in LLVM 2.4
Consider four target architectures and two code quality metrics

- x86
- x86-64
- Thumb
- ARM

More registers
Fewer registers
Limitations

• Self-selecting bias in results
  – limited to those functions where an optimal solution can be found in reasonable timeframe
  – however, qualitative results do not appear to change as more time is allowed for optimal allocator

• Implement swap using memory location

• Performance metric necessarily inexact (weighted sum of memory operations)

• Evaluate performance only on desktop processors
Results: Code Size

• Evaluate subset of Mibench
• Consider all functions where optimal solutions can be found in <10 minutes
  – more than 70% coverage of functions
• Report code size increase relative to fully optimal (1.0 best possible result)
Results: Code Performance

- Evaluate subset of SPEC2006
- Optimize only critical (>85% of running time) functions
- Intel Core 2 Quad (Q6600) @ 2.4GHz
- Report geometric mean relative to fully optimal model
- Possible to do better than optimal due to limitations of metric
Move Insertion: Code Size

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Code Size Relative to Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>1.0025</td>
</tr>
<tr>
<td>x86-64</td>
<td>1.002</td>
</tr>
<tr>
<td>ARM</td>
<td>1.0015</td>
</tr>
<tr>
<td>Thumb</td>
<td>1.001</td>
</tr>
</tbody>
</table>

- **No Move Insertion**
- **Limited Move Insertion**
Move Insertion: Code Performance

Relate to Optimal Allocation

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Loads</th>
<th>Stores</th>
<th>Instructions</th>
<th>Time</th>
<th>Loads</th>
<th>Stores</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x86-64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- No Move Insertion
- Limited Move Insertion
Coalescing: Code Size

No Coalescing
Separate Optimal Coalescing
Separate Aggressive Coalescing
Integrated Coalescing - Ignore Uncoalescables
Coalescing: Code Performance

![Graph showing performance relative to optimal allocation for different coalescing strategies (No Coalescing, Separate Optimal Coalescing, Separate Aggressive Coalescing, Integrated Coalescing - Ignore Uncoalescables) for x86 and x86-64 architectures. The graph uses a bar chart with relative time, loads, stores, and instructions as the x-axis, and relative to optimal allocation on the y-axis.]
Spilling: Code Size

![Bar chart comparing code size relative to optimal for x86, x86-64, ARM, and Thumb.

- **x86**: Separate Optimal Spilling
  - Code Size Relative to Optimal: 1.015
- **x86-64**: Separate Optimal Spilling
  - Code Size Relative to Optimal: 1.005
- **ARM**: Separate Heuristic Spilling
  - Code Size Relative to Optimal: 1.02
- **Thumb**: Separate Heuristic Spilling
  - Code Size Relative to Optimal: 1.03]
Spilling: Code Performance

![Graph showing performance of x86 and x86-64 architectures with separate optimal and heuristic spilling]

Relative to Optimal Allocation

Time | Loads | Stores | Instructions | Time | Loads | Stores | Instructions
---|---|---|---|---|---|---|---
x86 | | | | x86-64 | | | |

- Separate Optimal Spilling
- Separate Heuristic Spilling
Assignment: Code Size

<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th>x86-64</th>
<th>ARM</th>
<th>Thumb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size Relative to Optimal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.04</td>
<td>1.08</td>
<td>1.14</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Legend:
- Blue: Separate Optimal Assignment
- Red: Separate Heuristic Assignment - Scan Based
- Green: Separate Heuristic Assignment - Graph Based
### Assignment: Code Performance

<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th>x86-64</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.95</td>
<td>0.97</td>
<td>0.99</td>
</tr>
<tr>
<td>Loads</td>
<td>0.97</td>
<td>0.99</td>
<td>1.01</td>
</tr>
<tr>
<td>Stores</td>
<td>1.01</td>
<td>1.03</td>
<td>1.05</td>
</tr>
<tr>
<td>Instruc9ons</td>
<td>1.03</td>
<td>1.05</td>
<td>1.07</td>
</tr>
</tbody>
</table>

**Relative to Optimal Allocator**

- **Separate Optimal Assignment**
- **Separate Heuristic Assignment - Scan Based**
- **Separate Heuristic Assignment - Graph Based**
Heuristics: Code Size

Code Size Relative to Optimal

- Separate Heuristic Spilling and Assignment (Scan)
- Separate Heuristic Spilling and Assignment (Graph)
- Integrated Heuristic Spilling and Assignment

Platforms:
- x86
- x86-64
- ARM
- Thumb
### Heuristics: Code Performance

<table>
<thead>
<tr>
<th>Relative to Optimal Allocation</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **x86**: Separate Heuristic Spilling and Assignment (Scan), Separate Heuristic Spilling and Assignment (Graph), Integrated Heuristic Spilling and Assignment
- **x86-64**: Separate Heuristic Spilling and Assignment (Scan), Separate Heuristic Spilling and Assignment (Graph), Integrated Heuristic Spilling and Assignment
Conclusions

• When targeting processor **performance**, new register allocator designs should focus on solving **spill code optimization** as the coalescing, move insertion, and register assignment problems are adequately solved using existing heuristics.

• When targeting **code size**, new register allocator designs should focus on solving both the **spill code optimization and register assignment problems**, possibly in an **integrated** framework.