Register Allocation for Irregular Architectures

David Koes
CALCM
4/20/2004
Motivation

Good register allocation critical for performance

int i, j, k;
short a, b, c;
Outline

- Register Allocation Overview
- ...for Irregular Architectures
- Previous Work
- A New Hope?
Example

```
v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
  <- x
  <- w
  <- t
  <- u
```

*need to assign a register to hold the value of each variable*
Register Allocation

- For fixed number of registers
  - does an assignment exist?
  - if not, what should be spilled (later...)
- Find the assignment

When can we not assign two variables to the same register?
Liveness

- A variable is **live** at a point if the variable might be used later in the program
- Variables live at the same point:
  - cannot be in the same register
  - interfere
Interference

\begin{align*}
v & \leftarrow 1 \\
w & \leftarrow v + 3 \\
x & \leftarrow w + v \\
u & \leftarrow v \\
t & \leftarrow u + x \\
& \leftarrow x \\
& \leftarrow w \\
& \leftarrow t \\
& \leftarrow u
\end{align*}

Variables that are live at the same point interfere

How to represent this?
Interference

Variables that are live at the same point interfere

How to represent this?
Interference Graph

Variable → Node
Interference → Edge

Register Assignment →
Graph Coloring

\[
\begin{align*}
v & \leftarrow 1 \\
w & \leftarrow v + 3 \\
x & \leftarrow w + v \\
u & \leftarrow v \\
t & \leftarrow u + x \\
x & \leftarrow x \\
w & \leftarrow w \\
t & \leftarrow t \\
u & \leftarrow u
\end{align*}
\]
Graph Coloring

Given \( k \) colors, is it possible to color the nodes of a graph such that a node does not have the same color as any of its neighbors?

Register \( \rightarrow \) Color
Graph Coloring

Given $k$ colors, is it possible to color the nodes of a graph such that a node does not have the same color as any of its neighbors?

Register → Color
Graph Coloring Register Allocation

- Compute liveness information
- Build interference graph
- Use heuristics to color graph
  - use local properties like node degree
  - suboptimal: commit to coloring decisions
- Coloring succeeds – done
- Coloring fails – must spill
  - spilling “removes” variable from graph
Spilling

\[
\begin{align*}
v & \leftarrow 1 \\
w & \leftarrow v + 3 \\
x & \leftarrow w + v \\
u & \leftarrow v \\
t & \leftarrow u + x \\
x & \leftarrow x \\
w & \leftarrow w \\
t & \leftarrow t \\
u & \leftarrow u \\
k & = 3
\end{align*}
\]

Impossible to color
Spill a variable
– allocate to memory
– pick using heuristic
\begin{align*}
v &\leftarrow 1 \\
w_1 &\leftarrow v + 3 \\
M_w &\leftarrow w_1 \\
w_2 &\leftarrow M_w \\
x &\leftarrow w_2 + v \\
u &\leftarrow v \\
t &\leftarrow u + x \\
x &\leftarrow x \\
w_3 &\leftarrow M_w \\
 &\leftarrow w_3 \\
 &\leftarrow t \\
 &\leftarrow u
\end{align*}
Register Allocation Summary

- Build interference graph
- Color using heuristics
- If not colorable, spill
  - repeat process [Briggs 94]
  - single pass
    - all uncolored variables spilled
    - much faster compile time
Outline

• Register Allocation Overview
• ...for Irregular Architectures
• Previous Work
• A New Hope
Irregular Architectures

- Few registers
- Register usage restrictions
  - address registers, hardwired registers...
- Memory operands
- Examples:
  - x86, 68k, ColdFire, ARM Thumb, MIPS16, V800, various DSPs...
Register Allocation for Irregular Architectures

• Graph coloring register allocation used
  • gcc, ORC, SUIF, GHS, IMPACT, IBM

• Assertion:
  • Graph coloring is the wrong representation for performing register allocation on irregular architectures
Fewer Registers → More Spills

- Used gcc to compile >10,000 functions from Mediabench, Spec95, Spec2000, and micro-benchmarks
- Recorded for which functions graph coloring failed
PPC (32 registers)
68k (16 registers)
x86 (8 registers)

Increase in Spills as Number of Variables in Function Grows

Number of Functions

Number of Variables in Function
Graph Coloring and Spills

- Graph coloring solves the register sufficiency problem
  - Even if P=NP, suboptimal if spills necessary
- No optimization of spill code
- Many spills may slow down allocator
  - has to rebuild interference graph
Register Usage Restrictions

- Example

68k

MOVE (ptr),tmp1
EOR #3,tmp1
MOVEQ #32,tmp2 or MOVEA #32,tmp2
ADD tmp2,ptr
MOVE tmp1,D0

<table>
<thead>
<tr>
<th>A0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>D1</td>
</tr>
<tr>
<td>A2</td>
<td>D2</td>
</tr>
<tr>
<td>A3</td>
<td>D3</td>
</tr>
<tr>
<td>A4</td>
<td>D4</td>
</tr>
<tr>
<td>A5</td>
<td>D5</td>
</tr>
<tr>
<td>A6</td>
<td>D6</td>
</tr>
<tr>
<td>A7</td>
<td>D7</td>
</tr>
</tbody>
</table>
Register Usage Restrictions

- Instructions may require or prefer a specific subset of registers
  - 68k address/data registers
    - MOVEA #1,A0 // 4 byte instruction
    - MOVEQ #1,D0 // 2 byte instruction
  - x86 div instruction
- Graph coloring assumes all colors are equally applicable
  - no principled way to express preferences
  - requirements may be mutually exclusive
Memory Operands

• A variable allocated to memory may not require load/store to access
  ◦ depends upon instruction
  ◦ still less efficient than register access

• Graph coloring (usually) spills variables which make graph easier to color
  ◦ may not be an efficient variable to spill
Graph Coloring Wrong Approach for Irregular Architectures

• Solves wrong problem
  ◆ focuses attention on preventing spills
  ◆ doesn’t optimize spill code
• No representation of irregular features
• Variables assigned single register
  ◆ complicates meeting usage restrictions
  ◆ live range splitting partial solution
Outline

• Register Allocation Overview
• ...for Irregular Architectures
• Previous Work
   Graph coloring improvements
   Integer Programming
   Separated IP
   PBQP
• A New Hope
Graph Coloring Improvements

• Spill code optimization
  ● better heuristics [Bernstein et al 89]
  ● partial spilling [Bergner et al 97]

• Register usage constraints
  ● modified interference graph [Briggs 92]
  ● weighted interference graph/modified heuristics [Smith and Holloway 01]
Outline

• Register Allocation Overview
• ...for Irregular Architectures
• Previous Work
  ♦ Graph coloring improvements
  ♦ Integer Programming
  ♦ Separated IP
  ♦ PBQP
• A New Hope
Integer Programming (IP)

- Minimize/maximize linear function
- Subject to linear constraints
- Solution must be integer
- Example

Maximize \( z = x_1 + x_2 \)
subject to
\[ 2x_1 + 3x_2 \leq 12, \]
\[ x_1 \leq 4, \quad x_2 \leq 3 \]

Solution:
\[ x_1 = 4, \quad x_2 = 1 \]
\[ z = 5 \]
Register Allocation as IP

• Simplified example

\[
\begin{align*}
    a & \leftarrow \\
    b & \leftarrow \\
    c & \leftarrow a + b \\
    & \leftarrow a + c \\
    & \leftarrow b
\end{align*}
\]

\[
\begin{align*}
    \min \sum & 3m_a + 3m_b + 2m_c \\
    \text{subject to} & \quad m_a + m_b + m_c \geq 1 \\
    & \quad 0 \leq m_a, m_b, m_c \leq 1
\end{align*}
\]

\(m_{\text{var}}\) is a decision variable

0 means var is in register

1 means var is in memory
IP: Good News

• IP can precisely model register allocation [Goodwin and Wilken 96]
  ◆ including irregular architecture features [Kong and Wilken 98]
  ◆ can exploit structure of register allocation problem to improve compile time [Fu and Wilken 2002]

• Can solve problem without integer conditions in polynomial time
**IP: Bad News**

- With integer conditions problem is NP-complete
- No polynomial guarantee
- Does not get **feasible** solution quickly
  - can’t just impose time limits and get a usable, if suboptimal, solution
IP: Results

- SPEC92 (integer)
- x86, models many irregular features
- 61% reduction in runtime spill code overhead
- >15 minutes on 2.4% of SPEC92 functions

T. Kong and K. Wilken, “Precise Register Allocation for Irregular Architectures” 1998
Outline

- Register Allocation Overview
- ...for Irregular Architectures
- Previous Work
  - Graph coloring improvements
  - Integer Programming
  - Separated IP
  - PBQP
- A New Hope
Separated IP

- Separate allocation and assignment [Appel and George 01]
- Use IP to optimally insert spill code
  - also model some x86 features
- Result never has more than k live variables at any point
  - not necessarily k–colorable
  - insert moves at every program point
Separated IP: Second Pass

- Second pass performs assignment and removes moves
  - use heuristic solution [Park and Moon 98]
  - optimal solution (IP) not tractable
  - left as an open problem in paper
Separated IP: Results


Overall 9.5% improvement in execution speed
Separated IP: Limitations

- Can still be prone to exponential blow-up in first pass
  - may not provide intermediate solution
- Second pass not optimal
- Claims to be faster than full IP solution
  - different compilers, benchmarks, source languages, and target architectures
Outline

• Register Allocation Overview
• ...for Irregular Architectures
• Previous Work
   Graph coloring improvements
   Integer Programming
   Separated IP
   PBQP
• A New Hope
Partitioned Boolean Quadratic Optimization Problem Formulation

- Similar to IP [Scholz and Eckstein 02]
  - minimize quadratic function
  - decision variables 0–1
  - constraints incorporated into function

\[
f = \sum_{1 \leq i < j \leq n} x_i^T C_{ij} x_j + \sum_{1 \leq i \leq n} x_i^T c_i \rightarrow \min
\]

s.t.
\[
\begin{align*}
  x_i & \in \{0, 1\}^{|c_i|} & \forall 1 \leq i \leq n \\
  x_i^T \mathbf{1} & = 1
\end{align*}
\]
Partitioned Boolean Quadratic Optimization Problem Formulation

• Advantages
  ♦ Can fully model irregular features
  ♦ Fast, polynomial approximation performs well in practice

• Disadvantages
  ♦ Approximation algorithm not bounded
  ♦ No iterative way to improve upon solution
PBQP: Results

- Caramel 20xx DSP
  - very irregular register requirements
- Geometric mean improvement
  - Optimal: 5.85%
  - Approximation: 3.93%

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution Time</th>
<th>Improvement %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>oPBQP</td>
<td>hPBQP</td>
</tr>
<tr>
<td>biq</td>
<td>157808</td>
<td>164977</td>
</tr>
<tr>
<td>fft</td>
<td>80099</td>
<td>83399</td>
</tr>
<tr>
<td>hdvd</td>
<td>23370</td>
<td>23370</td>
</tr>
<tr>
<td>mmult</td>
<td>8165</td>
<td>8165</td>
</tr>
<tr>
<td>vit</td>
<td>194316</td>
<td>195671</td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Optimize spills code</th>
<th>Models irregular features</th>
<th>Polynomial running time</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Coloring</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Integer Programming</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Separated IP</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>PBQP</td>
<td>yes</td>
<td>yes</td>
<td>yes/no</td>
<td>no/yes</td>
</tr>
</tbody>
</table>
Outline

• Register Allocation Overview
• ...for Irregular Architectures
• Previous Work
  ♦ Graph coloring improvements
  ♦ Integer Programming
  ♦ Separated IP
  ♦ PBQP
• A New Hope
New Problem Formulation

Goals

• Explicitly represent architectural irregularities and costs
• An optimum solution results in optimal register allocation
• Suboptimal solution algorithm scales
  ♦ more computation → better solution
  ω decent feasible solution obtained quickly
  ♣ competitive with current allocators
One Possibility: Multicommodity Network Flow

• Given network (directed graph) with
  ♦ cost and capacity on each edge
  ♦ sources & sinks for multiple commodities
• Find lowest cost flow of commodities
• Many different applications
  ♦ communication networks, transportation networks, distribution networks, etc
• NP-complete for integer flows
MCNF: Example

Thin edges have capacity of one
Thick edges have infinite capacity
Cost is zero unless labeled
MCNF: Example

Thin edges have capacity of one
Thick edges have infinite capacity
Cost is zero unless labeled
Register Allocation as MCNF

- Variables → Commodities
- Variable Usage → Network Design
- Registers Limits → Bundle Constraints
- Spill Costs → Edge Costs
- Variable Definition → Source
- Variable Last Use → Sink
Example

```c
int foo(int a, int b)
{
    int c = a - b;
    return c / b;
}
```
MCNF Representation

• Explicitly optimizes spill code, memory operands, and register preferences
  ♦ represented by edge costs
• Most restrictions on register usage easily modeled
  ♦ capacity and bundle constraints
• Compact representation
MCNF as Integer Program

Minimize $\sum_k c^k x^k$

subject to

$\sum_k x^k_{ij} \leq u_{ij}$

$N x^k = b^k$

$0 \leq x^k_{ij} \leq u^k_{ij}$

- Variable for every commodity for every edge
  - flow of that commodity along that edge
- Flow constraints
  - bundle
  - network
  - capacity
Solving an MCNF

• Can use standard IP solvers
• Can exploit structure of problem
  - variety of MCNF specific solvers
    - empirically faster than IP solvers
    - integer solution still worst case exponential
• Noninteger solutions used to get integer solution
  - used to reduce search space
    - branch and bound
    - branch and cut
Lagrangian Relaxation

- Bring constraints into min function

\[ L(w) = \min \sum_{k} c^{k} x^{k} + \sum_{i,j} w_{ij} \left( \sum_{k} x^{k}_{ij} - u_{ij} \right) \]

\[ L(w) = \min \sum \sum_{i,j} \left( c^{k}_{ij} + w_{ij} \right) x^{k}_{ij} - \sum_{i,j} w_{ij} u_{ij} \]

- Lagrangian multipliers: edge price
  - subgradient optimization finds optimal price

- Relaxation removes bundle constraints
Lagrangian Relaxation

- Bring constraints into min function

\[
L(w) = \min \sum_{k} c^{k} x^{k} + \sum_{i,j} w_{ij} \left( \sum_{k} x_{ij}^{k} - u_{ij} \right)
\]

- Lagrangian multipliers: edge price
  - subgradient optimization finds optimal price
- Relaxation removes bundle constraints
Heuristic Solution

- Iterate
  - solve independent single commodity network flows in Lagrangian relaxation
  - update Lagrangian multipliers
- Converge (or terminate at cutoff)
- Use prices to guide greedy algorithm
  - build solution from single commodity flow subproblems
Summary

• Graph coloring wrong approach for irregular architectures
• Other approaches
  ◆ can fully model architecture
  ◆ often optimal
  ◆ no performance guarantee
• Multicommodity network flow
  ◆ promising new formulation
Questions?
k-live, but not k-colorable
k-live, but not k-colorable
$k$-live, but not $k$-colorable
k-live, but not k-colorable
k-live, but not k-colorable