Progressive Register Allocation for Irregular Architectures

David Koes
2/1/2005

Outline

• Register Allocation for Irregular Architectures
• Previous Work
• Progressive Register Allocation

Irregular Architectures

• Few registers
• Register usage restrictions
  – address registers, hardwired registers...
• Memory operands
• Examples:
  – x86, 68k, ColdFire, ARM Thumb, MIPS16, V800, various DSPs...

Irregular Architectures Register Allocation

• 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

<table>
<thead>
<tr>
<th>Percent of functions with no spills</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC (32)</td>
</tr>
<tr>
<td>68k (16)</td>
</tr>
<tr>
<td>x86 (8)</td>
</tr>
</tbody>
</table>

PPC (32 registers)

Increase in Spills as Number of Variables in Function Grows

68k (16 registers)

Increase in Spills as Number of Variables in Function Grows

x86 (8 registers)

Increase in Spills as Number of Variables in Function Grows
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
```

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

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 for Irregular Architectures
- Previous Work
  - Graph coloring improvements
  - Integer Programming
  - Separated IP
  - PBQP
- Progressive Register Allocation

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]
Integer Programming (IP)

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

Example

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

Solution:

\[ x_1 = 4, x_2 = 1 \]
\[ z = 5 \]

Register Allocation as IP

- Simplified example

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

\( m_{\text{var}} \) 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

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

A. W. Appel, L. George. "Optimal spilling for CISC machines with few registers."

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
  - Irregular Architectures
- Previous Work
  - Graph coloring improvements
  - Integer Programming
  - Separated IP
  - PBQP
- Progressive Register Allocation

Partitioned Boolean Quadratic Optimization Problem Formulation

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

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

\[
\text{s.t.} \\
z_i \in \{0, 1\}^{|i|} \\
z_i^T 1 = 1 \\
\forall 1 \leq i \leq n
\]
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>mcf</td>
<td>15786</td>
<td>17000</td>
</tr>
<tr>
<td>bft</td>
<td>88009</td>
<td>83399</td>
</tr>
<tr>
<td>b harmed</td>
<td>23370</td>
<td>23370</td>
</tr>
<tr>
<td>crn</td>
<td>8155</td>
<td>8155</td>
</tr>
<tr>
<td>vit</td>
<td>194516</td>
<td>190671</td>
</tr>
</tbody>
</table>

Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Optimizes spill code</th>
<th>Models irregular features</th>
<th>Polynomial running time</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Coloring</td>
<td>with heuristics</td>
<td>some, with heuristics</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Integer Programming</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Separated IP</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>PBQP</td>
<td>yes</td>
<td>yes/no</td>
<td>no/yes</td>
<td></td>
</tr>
</tbody>
</table>

Outline

- Register Allocation
  - Irregular Architectures
- Previous Work
  - Progressive Register Allocation
    - MCNF Formulation
    - Solution Procedure
    - Results
Problem Formulation Goals

• Explicitly represent architectural irregularities and costs
• An optimum solution results in optimal register allocation
• Solution algorithm is progressive
  – more computation → better solution
  – decent feasible solution obtained quickly
  – competitive with current allocators

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

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

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

Outline

- Register Allocation
- Irregular Architectures
- Previous Work
- Progressive Register Allocation
  - MCNF Formulation
  - Solution Procedure
  - Results

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 bundle constraints into min function

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

- Lagrangian multipliers \( w_{ij} \); edge price
- For given \( w \), computing \( L(w) \) is simple shortest path computation

Solution Procedure

- Compute \( L(w) \)
  - lower bound of optimal value
- Update price vector \( w \)
  - take a step in the direction of the gradient
  - guaranteed to converge (eventually)
- Use prices to guide greedy algorithm
  - build solution from single path subproblems
  - can always find feasible solution
- Repeat
Solution Procedure

• Advantages
  – have feasible solution at each step
  – can compute optimality guarantee
    • from lower bound
  – iterative nature → progressive

• Disadvantages
  – No guarantee of optimality of solution

Outline

• Register Allocation
  Irregular Architectures
• Previous Work
• Progressive Register Allocation
  – MCNF Formulation
  – Solution Procedure
  – Results

Evaluation

• Replace gcc’s local allocator
• Optimize for code size
  – easy to statically evaluate
• Evaluate on MediaBench, MiBench, SpecInt95, SpecInt2000
  – consider only blocks where local allocation is interesting (enough variables to spill)

Behavior of Solver
Comprehensive Results

Optimality

Progressive Nature

Future Work

- Improve convergence of $L(w)$
- Better feasible solution finder
- Global register allocation