Mapping Reference Code to Irregular DSPs within the Retargetable, Optimizing Compiler COGEN(t)

Gary Grewal and Tom Wilson
Dept. of Computing and Information Science
University of Guelph
Guelph, Ontario, Canada
Outline

1. The problem of generating code for highly encoded instruction sets: to optimize or retarget?
2. Our approach: make optimization more generic by ignoring some details until near the end
3. The idealized version of the processor (clean machine)
4. Quick tour of how we compile for the clean machine
5. Shake And Bake: the final mapping to the real processor
6. Inside Shake and Bake: enhanced genetic algorithms
7. Closing comments
The Problem with Highly Encoded Instruction Sets

- Code generation becomes more difficult with…
  - Instruction level parallelism (especially if constrained)
  - Non-orthogonal structure (overlapping roles and special cases)
  - Small, heterogeneous register classes
  - Instruction-based constraints on allowable register sets
  - Instruction-based interdependencies among registers
- Specialized processors have a niche
- Optimized code is usually required for their applications
- Retargetability is important during design exploration
  - Trying different existing processors
  - Developing an ASIP or extending an existing processor family
Typical DSP Architecture

General DSP Architecture showing Register Classes:
- Accumulators
- Operand Registers
- Address Registers

Dual Data Memories and Register-to-Register ALU operations
To Optimize or Retarget?

- Optimizing for a “difficult” instruction set often involves clever, specialized techniques that don’t adapt easily.
- Retargeting requires agile algorithms that often fail to find good enough code.
- The COGEN(t) Strategy:
  1. Optimize for a well-behaved target processor that resembles the real one – minus some ugly constraints
  2. Then map from the resulting code to code that satisfies all the constraints of the real processor
Sample Loop Kernel

double FIR_filter(in double A[], in double B[], in int tap) {
    int k;
    double sum=0;
    for(k=0; k< tap; k++)
        sum += A[k] * B[k];
    return sum;
}

<table>
<thead>
<tr>
<th>ALU</th>
<th>X-Memory</th>
<th>Y-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>A</td>
<td>X:(R0)+, X0</td>
</tr>
<tr>
<td>REP</td>
<td>#N-1</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>X0, Y0, A</td>
<td>X:(R0)+, X0</td>
</tr>
<tr>
<td>MACR</td>
<td>X0, Y0, A</td>
<td>Y:(R4)+, Y0</td>
</tr>
</tbody>
</table>

[1] [2] [3] [4]
Parallel Instruction Template

MAC        X0,Y0, A  X:(R0)+, X0  Y:(R4)+, Y0

Input Ports

Output Ports
Domains relate to specific units of the processor

MAC  X0, Y0, A  X:(R0)+, X0  Y:(R4)+, Y0
Domains and Allowable Register Sets
Register-to-Port Constraints

++
ld
macc
R4, R5, R6, R7
R4, R5, R6, R7
ld
R4, R5, R6, R7
++

A, B, Y0, Y1, R0, R1, R2, R3
R4, R5, R6, R7

R4, R5, R6, R7
MAC  X0, Y0, A  X: (R0)+, X0  Y: (R4)+, Y0
Effect of Parallelism on Constraints

Instruction Template

Register Sets

Constraints

\( \{R0, A0\} \text{ incompatible} \)  
\( \{R1, A1\} \text{ incompatible:} \)  
\( \{A0, A1\} \text{ incompatible} \)

\( P4 = P2 \) (same register)

if\( (P3 = L0) \) then  \( P5 = A0 \)  
if\( (P3 = L1) \) then  \( P5 = A1 \)  
\( P1 = P6 \) (same register)
Clean Machine – operator parallelism

- Each domain supports certain basic ops
- Ops from different domains may be used in parallel without restriction

Alignment of basic operations
Clean Machine – register sets

- Each basic operation in a domain has a set of allowable registers for each port.
- These register sets are unaffected by parallelism (they remain the same whether the op is isolated or used in parallel).
- Register sets that map to ports of the same instruction have no special external constraints between them.
Sample Loop Kernel – many connected instructions

double FIR_filter(in double A[], in double B[], in int tap) {
    int k;
    double sum=0;
    for(k=0; k< tap; k++)
        sum += A[k] * B[k];
    return sum;
}

<table>
<thead>
<tr>
<th>ALU</th>
<th>X-Memory</th>
<th>Y-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>A</td>
<td>X:(R0)+, X0</td>
</tr>
<tr>
<td>REP</td>
<td>#N-1</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>X0,Y0, A</td>
<td>X:(R0)+, X0</td>
</tr>
<tr>
<td>MACR</td>
<td>X0, Y0, A</td>
<td></td>
</tr>
</tbody>
</table>
The COGEN(t) Retargetable Code Generator

- Patterns map basic ops into supported ops, restructure
- Basic blocks partitioned into traces, inner loops first
- Schedule mainstream ops and bind ops to domains using an “enhanced genetic algorithm” (EGA)
- Spills introduced to meet register availability
- Memory-resident values mapped to final memory
- Address registers and address update code inserted
- Relational Database supplies intermediate storage ➔
  - powerful search capability
  - easy to maintain pools of alternatives

Note: scheduler finds “optimum” schedule by successive discovery of “feasible” length-constrained schedules
Pool Methodology

adjust for:
• architecture
• type of trace

relative importance of factors

Candidate pool – all feasible

EGA

Criteria

max

Others may be kept as backups
Reference Code produced by COGEN(t)

- In every trace, the code observes constraints:
  - At most one basic op per domain per instruction (1)
  - The assignment of ops to domains is appropriate (2)
  - All necessary orderings are observed (3)

  Note: all parallel domain combinations (alignments) are allowed in the clean machine

- In every trace, the code consumes the fewest possible steps and is as parallel as possible

- At every step of every trace, the number of live variables does not exceed the number of suitable registers

  Note: Actual registers are assigned later by Bake
No Register Conflicts
Task of *Shake And Bake*

- **General objective:** Map reference code to actual target instruction set
- **Specifically:**
  - Perturb reference code, so that alignments of basic operations all correspond to actual machine instructions
  - Find an actual register assignment to all live variables, so that encoding constraints of target instruction set are observed
Overview of *Shake And Bake*

- **SHAKE** generates a pool of alternative reference code sequences, equivalent to the original reference code, but with basic ops aligned differently.
- **AND** verifies that every alignment in the reference code corresponds to an actual (parallel) instruction.
- **AND** inserts *register copy* operations on edges having no feasible register assignment.
- **BAKE** discovers (if possible) a feasible register assignment meeting all constraints.
Action of SHAKE: produce pool of candidate alignments

Action of AND: map aligned operations to real instructions
Effect of Different Alignments

{ } - possible register assignment
The effect of relaxing parallelism

(a)                               (b)                           (c)                                (d)

register mismatches
AND verifies instructions (feasible alignments) and determines register sets for each connecting edge.
AND: register *copy* operations resolve mismatches
Operation of BAKE

- BAKE uses an Enhanced Genetic Algorithm (EGA) to “shower” the edges of reference code, until a feasible register assignment is found
- Observes many constraints:
  - Every edge is assigned a register from a suitable class
  - Restrictions imposed by parallelism: each instruction port can be serviced only by suitable register subclass
  - Inter-instruction: same register used throughout the life of any edge, with no conflicting uses
  - Intra-instruction: explicit “same/different”, “if-then”, and “iff” register assignment at different ports
Interactions Between Instructions – problem for BAKE

Dependencies between ports may restrict allowable registers

Disjoint
Combined Effect of Shake And Bake

SHAKE: Find new Alignments

BAKE: Register Assignment

AND: Valid instructions
AND: Copy insertion

SELECT

Object Code
Flow though Shake and Bake
A genetic algorithm (GA) is depicted with its components and operations. The genetic algorithm involves a pool of candidate solutions that undergo selection, crossover with a large probability, and mutation with a small probability. The fitness of each candidate solution, denoted as $f_i$, influences the selection process. Crossover and mutation result in a candidate pool for the next generation. The diagram also illustrates order constraints and step constraints, with edges connecting between registers and many reg-port constraints. The genetic algorithm aims to optimize a solution through these processes.
**GA**

- Fitness $f_i$
- Fitness $f_j$

**EGA**

- Fitness $f_i$
- Fitness $f_j$

- Crossover always performed
- Select best two (based on fewest violated constraints)
- Always mutate violated constraint
  -- alter a variable in each violated constraint
Results for Biquad Filter

• Results generated by SAB at each step of the mapping process

<table>
<thead>
<tr>
<th>Patterns?</th>
<th>DSP</th>
<th>SHAKE Alignments</th>
<th>AND Feasible</th>
<th>AND Copies</th>
<th>BAKE Feasible</th>
<th>Steps in Schedule</th>
<th>Time per schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>M56K</td>
<td>30</td>
<td>29</td>
<td>0</td>
<td>29</td>
<td>11</td>
<td>3.0s</td>
</tr>
<tr>
<td>No</td>
<td>ST950</td>
<td>30</td>
<td>21</td>
<td>2-5</td>
<td>15</td>
<td>14</td>
<td>3.1s</td>
</tr>
<tr>
<td>Yes</td>
<td>M56K</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>29</td>
<td>8</td>
<td>2.0s</td>
</tr>
<tr>
<td>Yes</td>
<td>ST950</td>
<td>30</td>
<td>30</td>
<td>2</td>
<td>30</td>
<td>11</td>
<td>4.3s</td>
</tr>
</tbody>
</table>
Mapping Code to DSPs and ASIPs

- ASIPs differ in register number, type, and connectivity [Leupers 97]

<table>
<thead>
<tr>
<th></th>
<th>M56K</th>
<th>ST950</th>
<th>ASIP1</th>
<th>ASIP2</th>
<th>ASIP3</th>
<th>ASIP4</th>
<th>ASIP5</th>
<th>ASIP6</th>
<th>ASIP7</th>
<th>ASIP8</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>8</td>
<td>11</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>T2</td>
<td>7</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>T3</td>
<td>16</td>
<td>27</td>
<td>17</td>
<td>16</td>
<td>16</td>
<td>20</td>
<td>16</td>
<td>16</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>
Shake and Bake in Other Contexts

- Instruction Set Capture
- Clean Machine
- Real Machine
- ISS Generator
- Clean ISS
- Real ISS
- Source Program
- Compile
- Reference Code
- Shake & Bake
- Object Code