Lecture 16: Embedded Compiler Optimisations

Michael O’Boyle
Embedded Software
Overview

- Introduction
- Address generating units
  - Single offset assignment problem
- Instruction level parallelism
  - Going beyond list scheduling
- Vectorisation for multimedia instructions
- Avoiding branch delay
  - Using predicated instructions
- Function inlining
  - Trade off between code size and performance
- Summary
Introduction

- Previously looked at large scale mapping issues
  - Look at smaller scale intra-processor utilisation
- Traditionally embedded processors programmed in assembler
  - As programmer cost rises, more emphasis on tools
- Compilers are a key component in exploiting embedded systems
- Different challenges and opportunities compared to general purpose
  - Architectures more complex and pre-specialised
  - Code size energy as well as speed
  - More work to do as less hardware support
- Compile time can take longer as amortised over number of uses
Address generation unit in parallel

Code: \( y_{i[j]} = y_{i-1[j]} + x[j-i]*a[i] \) for all \( i, j \)

Example: Data path ADSP210x

- Parallelism
- Dedicated registers
- No matching compiler ⇒ inefficient code

Address registers: A0, A1, A2 .. i+1, j-i-1

Address generation unit (AGU)
Compiling for AGus

- In effect does restricted register indirect addressing in parallel
  
  LD r31, Mem[r1]
  ADD r1, r1, 1
  LD r31, Mem[r1]

- Is replaced by
  
  LD r31, Mem[a1]; (a1++)

- Where the a1++ may be explicit or implicit

- Very useful for loops and array refs
  
  Eliminates an instruction per load - code size
  However a challenge on how to utilise agus
Handling array references in loops

Example:
for (i=2; i<=N; i++)
{ .. B[i+1] /*A2++ */
  .. B[i]    /*A1-- */
  .. B[i+2] /*A2++ */
  .. B[i-1] /*A1++ */
  .. B[i+3] /*A2-- */
  .. B[i] } /*A1++ */

Cost for crossing loop boundaries considered.

Reference: A. Basu, R. Leupers, P. Marwedel:
Array Index Allocation under Register Constraints,
Int. Conf. on VLSI Design, Goa/India, 1999
Exploitation of parallel address computations

Generic address generation unit (AGU) model

Parameters:
\( k = \# \text{ address registers} \)
\( m = \# \text{ modify registers} \)

Cost metric for AGU operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate AR load</td>
<td>1</td>
</tr>
<tr>
<td>immediate AR modify</td>
<td>1</td>
</tr>
<tr>
<td>auto-increment/ decrement</td>
<td>0</td>
</tr>
<tr>
<td>AR += MR</td>
<td>0</td>
</tr>
</tbody>
</table>
Let's assume that we can modify the memory layout offset assignment problem.

$(k,m,r)$-OA is the problem of generating a memory layout which minimizes the cost of addressing variables, with

- $k$: number of address registers
- $m$: number of modify registers
- $r$: the offset range

The case $(1,0,1)$ is called simple offset assignment (SOA), the case $(k,0,1)$ is called general offset assignment (GOA).
Variables in a basic block:  Access sequence:

\[ V = \{a, b, c, d\} \quad S = (b, d, a, c, d, c) \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load AR,1 ;b
AR += 2 ;d
AR -= 3 ;a
AR += 2 ;c
AR ++ ;d
AR -- ;c

cost: 4

SOA example
- Effect of optimised memory layout -
Variables in a basic block: $V = \{a, b, c, d\}$

Access sequence: $S = (b, d, a, c, d, c)$

<table>
<thead>
<tr>
<th>0</th>
<th>a</th>
<th>Load AR,1, ;b</th>
<th>0</th>
<th>b</th>
<th>Load AR,0, ;b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>AR += 2, ;d</td>
<td>1</td>
<td>d</td>
<td>AR ++, ;d</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>AR -= 3, ;a</td>
<td>2</td>
<td>c</td>
<td>AR +=2, ;a</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>AR += 2, ;c</td>
<td>3</td>
<td>a</td>
<td>AR --, ;c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AR ++,</td>
<td></td>
<td></td>
<td>AR --,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;d</td>
<td></td>
<td></td>
<td>;c</td>
</tr>
</tbody>
</table>

cost: 4  cost: 2
SOA example: Access sequence, access graph and Hamiltonian paths

access sequence: b d a c d c

[Bartley, 1992; Liao, 1995]
SOA example: Access sequence, access graph and Hamiltonian paths

- Access sequence: b d a c d c

- Access graph

[Bartley, 1992; Liao, 1995]
SOA example: Access sequence, access graph and Hamiltonian paths

access sequence: b d a c d c

access graph

maximum weighted path = max. weighted Hamilton path covering (MWHC)

[Bartley, 1992; Liao, 1995]
SOA example: Access sequence, access graph and Hamiltonian paths

access sequence: b d a c d c

access graph

maximum weighted path = max. weighted Hamilton path covering (MWHC)

memory layout

[Bartley, 1992; Liao, 1995]
SOA example: Access sequence, access graph and Hamiltonian paths

access sequence: b d a c d c

access graph

maximum weighted path = max. weighted Hamilton path covering (MWHC)

memory layout

SOA used as a building block for more complex situations

[Bartley, 1992; Liao, 1995]
SOA example: Access sequence, access graph and Hamiltonian paths

access sequence: b d a c d c

access graph

maximum weighted path = max. weighted Hamilton path covering (MWHC)

memory layout

SOA used as a building block for more complex situations

significant interest in good SOA algorithms

[Bartley, 1992; Liao, 1995]
Naïve SOA

Nodes are added in the order in which they are used in the program.

Example:
Access sequence: \( S = (b, d, a, c, d, c) \)
Naïve SOA

Nodes are added in the order in which they are used in the program.

Example:
Access sequence: \( S = (b, d, a, c, d, c) \)

\[
\begin{array}{|c|c|c|c|c|}
\hline
1 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
0 & b \\
\hline
1 & d \\
\hline
2 & a \\
\hline
3 & c \\
\hline
\end{array}
\]

memory layout
Naïve SOA

Nodes are added in the order in which they are used in the program.

Example:
Access sequence: \( S = (b, d, a, c, d, c) \)
Naïve SOA

Nodes are added in the order in which they are used in the program.

Example:
Access sequence: \( S = (b, d, a, c, d, c) \)
Naïve SOA

Nodes are added in the order in which they are used in the program.

Example:
Access sequence:  \( S = (b, d, a, c, d, c) \)

memory layout
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph \( G = (V, E) \) according to their weight
2. Construct a new graph \( G' = (V', E') \), starting with \( E' = 0 \)
3. Select an edge \( e \) of \( G \) of highest weight; If this edge does not cause a cycle in \( G' \) and does not cause any node in \( G' \) to have a degree > 2 then add this node to \( E' \) otherwise discard \( e \).
4. Goto 3 as long as not all edges from \( G \) have been selected and as long as \( G' \) has less than the maximum number of edges (\( |V| - 1 \)).

Example: Access sequence: \( S = (b, d, a, c, d, c) \)

Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph $G = (V, E)$ according to their weight
2. Construct a new graph $G' = (V', E')$, starting with $E' = 0$
3. Select an edge $e$ of $G$ of highest weight; If this edge does not cause a cycle in $G'$ and does not cause any node in $G'$ to have a degree > 2 then add this node to $E'$ otherwise discard $e$.
4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G'$ has less than the maximum number of edges ($|V| - 1$).

Example: Access sequence: $S = (b, d, a, c, d, c)$

Implicit edges of weight 0 for all unconnected nodes.
Liao's algorithm

Similar to Kruskal's spanning tree algorithms:
1. Sort edges of access graph $G=(V,E)$ according to their weight
2. Construct a new graph $G'=(V',E')$, starting with $E'=0$
3. Select an edge $e$ of $G$ of highest weight; If this edge does not cause a cycle in $G'$ and does not cause any node in $G'$ to have a degree $>2$ then add this node to $E'$ otherwise discard $e$.
4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G'$ has less than the maximum number of edges ($|V|-1$).

Example: Access sequence: $S=(b, d, a, c, d, c)$
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph $G=(V,E)$ according to their weight
2. Construct a new graph $G’=(V’,E’)$, starting with $E’=0$
3. Select an edge $e$ of $G$ of highest weight; If this edge does not cause a cycle in $G’$ and does not cause any node in $G’$ to have a degree > 2 then add this node to $E’$ otherwise discard $e$.
4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G’$ has less than the maximum number of edges ($|V|-1$).

Example: Access sequence: $S=(b, d, a, c, d, c)$
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph $G=(V,E)$ according to their weight
2. Construct a new graph $G’=(V’,E’)$, starting with $E’=0$
3. Select an edge $e$ of $G$ of highest weight; If this edge does not cause a cycle in $G’$ and does not cause any node in $G’$ to have a degree $> 2$ then add this node to $E’$ otherwise discard $e$. 
4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G’$ has less than the maximum number of edges ($|V|-1$).

Example: Access sequence: $S=(b, d, a, c, d, c)$

```
<table>
<thead>
<tr>
<th>Access</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b, d)</td>
<td>1</td>
</tr>
<tr>
<td>(a, c)</td>
<td>1</td>
</tr>
<tr>
<td>(a, d)</td>
<td>1</td>
</tr>
<tr>
<td>(b, d)</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph $G=(V,E)$ according to their weight
2. Construct a new graph $G’=(V’,E’)$, starting with $E’=0$
3. Select an edge $e$ of $G$ of highest weight; If this edge does not cause a cycle in $G’$ and does not cause any node in $G’$ to have a degree > 2 then add this node to $E’$ otherwise discard $e$.
4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G’$ has less than the maximum number of edges ($|V|-1$).

Example: Access sequence: $S=(b, d, a, c, d, c)$

Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph \( G=(V,E) \) according to their weight
2. Construct a new graph \( G’=(V’,E’) \), starting with \( E’=0 \)
3. Select an edge \( e \) of \( G \) of highest weight; If this edge does not cause a cycle in \( G’ \) and does not cause any node in \( G’ \) to have a degree > 2 then add this node to \( E’ \) otherwise discard \( e \).
4. Goto 3 as long as not all edges from \( G \) have been selected and as long as \( G’ \) has less than the maximum number of edges (\(|V|-1\)).

Example: Access sequence: \( S=(b, d, a, c, d, c) \)

* Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph \( G=(V,E) \) according to their weight
2. Construct a new graph \( G’=(V’,E’) \), starting with \( E’=0 \)
3. Select an edge \( e \) of \( G \) of highest weight; If this edge does not cause a cycle in \( G’ \) and does not cause any node in \( G’ \) to have a degree > 2 then add this node to \( E’ \) otherwise discard \( e \).
4. Goto 3 as long as not all edges from \( G \) have been selected and as long as \( G’ \) has less than the maximum number of edges \((|V|-1)\).

Example: Access sequence: \( S=(b, d, a, c, d, c) \)

### Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph $G = (V, E)$ according to their weight
2. Construct a new graph $G' = (V', E')$, starting with $E' = 0$
3. Select an edge $e$ of $G$ of highest weight; If this edge does not cause a cycle in $G'$ and does not cause any node in $G'$ to have a degree $> 2$ then add this node to $E'$ otherwise discard $e$.
4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G'$ has less than the maximum number of edges ($|V| - 1$).

Example: Access sequence: $S = (b, d, a, c, d, c)$

Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm

Similar to Kruskal’s spanning tree algorithms:
1. Sort edges of access graph \( G=(V,E) \) according to their weight
2. Construct a new graph \( G’=(V’,E’) \), starting with \( E’=0 \)
3. Select an edge \( e \) of \( G \) of highest weight; If this edge does not cause a cycle in \( G’ \) and does not cause any node in \( G’ \) to have a degree > 2 then add this node to \( E’ \) otherwise discard \( e \).
4. Goto 3 as long as not all edges from \( G \) have been selected and as long as \( G’ \) has less than the maximum number of edges (\(|V|-1\)).

Example: Access sequence: \( S=(b, d, a, c, d, c) \)

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
G: \\
\begin{array}{c}
\text{a} \\
\text{c} \\
\text{d} \\
\end{array}
\begin{array}{c}
\text{b} \\
1 \\
2 \\
1 \\
\end{array}
\begin{array}{c}
1 \\
1 \\
2 \\
1 \\
\end{array}
\]

\[
G’: \\
\begin{array}{c}
\text{a} \\
\text{c} \\
\text{d} \\
\end{array}
\begin{array}{c}
\text{b} \\
1 \\
1 \\
2 \\
\end{array}
\begin{array}{c}
1 \\
1 \\
2 \\
1 \\
\end{array}
\]

Implicit edges of weight 0 for all unconnected nodes.
Liao’s algorithm on a more complex graph

\[ abcdedefadadaadadafad \]

\[ G \]

\[ G' \]

Thursday, 13 March 2014
Liao’s algorithm on a more complex graph

G

G’
Liao's algorithm on a more complex graph

a b c d e f a d a d a c d f a d

G

G'

Thursday, 13 March 2014
Liao’s algorithm on a more complex graph

a b c d e f a d a d a c d f a d

\[ G \]

\[ G' \]
Liao’s algorithm on a more complex graph

\[a b c d e f a d a d a c d f a d\]
Liao’s algorithm on a more complex graph

a b c d e f a d a d a c d f a d

$G$

$G'$

Thursday, 13 March 2014
Exploitation of instruction level parallelism (ILP)

Several transfers in the same cycle:

Address registers A0, A1, A2 .. i+1, j-i+1

Address generation unit (AGU)
Exploiting ILP

• Normally looked at as a way if improving performance
• However on vliw and dsp architectures
  • Also helps reduce code size
• If the parallel units are not doing anything
  • then have to fill with a nop
Exploitation of instruction level parallelism (ILP)

1: \( MR := MR + (MX \times MY) \);
2: \( MX := D[A1] \);
3: \( MY := P[A2] \);
4: \( A1 := - \);
5: \( A2 := ++ \);
6: \( D[0] := MR \);

Normally tackled using dependence graph and list scheduling

- Modelling of possible parallelism using n-ary compatibility relation, e.g. \( \sim (1, 2, 3, 4, 5) \)
- Generation of integer programming (IP)-model (max. 50 statements/model)
- Using standard-IP-solver to solve equations
Exploitation of instruction level parallelism (ILP)

Results obtained through integer programming:

Code size reduction [%]

bassboost
dct
equalize
fir12
lattice2
pidctrl
adaptive2
adaptive1

[Leupers, EuroDAC96]

Compaction times: 2 .. 35 sec

Thursday, 13 March 2014
Exploitation of Multimedia Instructions

FOR $i := 0$ TO $n$ DO
   $a[i] = b[i] + c[i]$

FOR $i := 0$ STEP 4 TO $n$ DO
   $a[i] = b[i] + c[i]$
   $a[i+1] = b[i+1] + c[i+1]$
   $a[i+2] = b[i+2] + c[i+2]$
   $a[i+3] = b[i+3] + c[i+3]$

A form of limited vectorisation. Normally performed in code generation stage rather than at restructure stage.
Improvements for M3 DSP due to vectorization

- example
- n_real_updates
- lms
- dot_product_2
- dot_product_16

rel. number of cycles [%]

original code

vectorized code

application
Avoiding branch delay using predication

**Large branch delay penalty:**

<table>
<thead>
<tr>
<th>15 (TriMedia) bzw.</th>
<th>40 (C62xx) delay</th>
<th>slots</th>
</tr>
</thead>
</table>

Avoiding this penalty: **predicated execution:**

- **[c] instruction**
  - c=true: instruction executed
  - c=false: effectively NOOP

**Realisation of if-statements**

with conditional jumps or with **predicated execution**:

```plaintext
if (c)
    { a = x + y;
      b = x + z;
    }
else
    { a = x - y;
      b = x - z;
    }
```

**Cond. instructions:**

- `[c] ADD x,y,a`
- `|| [c] ADD x,z,b`
- `|| ![c] SUB  x,y,a`
- `|| ![c] SUB  x,z,b`

1 cycle

Thursday, 13 March 2014
Cost of implementation methods for IF-Statements

Sourcecode: if (c1) {t1; if (c2) t2}

No precondition (no enclosing IF or enclosing IFs implemented with cond. jumps)

1. Conditional jump:
   BNE c1, L;
   t1;
   L: ...

2. Conditional Instruction:
   [c1] t1

Precondition (enclosing IF not implemented with conditional jumps)

3. Conditional jump:
   [c1] c:=c2
   [~c1] c:=0
   BNE c, L;
   t2;
   L: ...

4. Conditional Instruction:
   [c1] c:=c2
   [~c1] c:=0
   [c] t2

Additional computations to compute effective condition c
Optimization for nested IF-statements

Goal: compute fastest implementation for all IF-statements

- Selection of fastest implementation for if-1 requires knowledge of how fast if-2 can be implemented.
- Execution time of if-2 depends on setup code, and, hence, also on how if 1 is implemented
- cyclic dependency!
Dynamic programming algorithm (phase 1)

For each if-statement compute 4 cost values:
- T1: cond. jump, no precondition
- T2: cond. instructions, no precondition
- T3: cond. jump, with precondition
- T4: cond. instructions, with precondition
Dynamic programming (phase 2)

No precondition for top-level IF-statement. Hence, comparison $T_1 < T_2$ suffices.

$T_1 < T_2$:
cond. branch faster, no precondition for nested IF-statements

$T_1 > T_2$:
cond. instructions faster, precondition for nested IF-statements
Results: TI C62xx

Runtimes (max) for 10 control-dominated examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Conditional jumps</th>
<th>Conditional instructions</th>
<th>Dynamic program.</th>
<th>Min (col. 2-5)</th>
<th>TI C compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>21</td>
<td>22</td>
<td>21</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>30</td>
<td>24</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
<td>173</td>
<td>49</td>
<td>49</td>
<td>51</td>
</tr>
<tr>
<td>8</td>
<td>39</td>
<td>244</td>
<td>30</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>28</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td>30</td>
<td>30</td>
<td>27</td>
<td>28</td>
</tr>
</tbody>
</table>

Average gain: 12%
Function inlining: advantages and limitations

**Advantage: low calling overhead**

```plaintext
Function sq(c:integer) return:integer;
begin
  return c*c
end;
```

```plaintext
a = sq(b);
```

```plaintext
push PC;
push b;
BRA sq;
pull R1;
mul R1,R1,R1;
pull R2;
push R1;
BRA (R2)+1;
pull R1;
ST R1,a;
```

**Limitations:**
- Not all functions are candidates.
- Code size explosion.
- Requires manual identification using ‘inline’ qualifier.

**Goal:**
- Controlled code size
- Automatic identification of suitable functions.
Results for GSM speech and channel encoder: #calls, #cycles (TI ‘C62xx)

33% speedup for 25% increase in code size.
# of cycles not a monotonically decreasing function of the code size!
Summary

• Address generating units
  • Single offset assignment problem
• Instruction level parallelism
  • Going beyond list scheduling
• Vectorisation for multimedia instructions
• Avoiding branch delay
  • Using predicated instructions
• Function inlining
  • Trade off between code size and performance