Compiler and Code Generation 2

SS 2009
Hw/Sw Codesign

Christian Plessl
Compiler and Code Generation

- Compiler structure, intermediate code
- Code generation
- Code optimization
- Code generation for specialized processors
- Retargetable compiler
Compilers for Embedded Processors

- Software for embedded systems
  - transition from assembly to high-level languages (HLLs)

- Main requirements: code has to be …
  - correct
  - fast
  - small
  - low power

- Further requirements on HLL and compiler
  - formal verification possible
  - specification of real-time constraints
  - support of DSP/multi-media algorithms and architectures
  - retargetable: quickly adaptable to new processors
Selected Problems

• Inhomogeneous register files, irregular data paths
  – tight phase coupling: register binding, instruction binding, instruction sequencing

• Allocation of memory addresses and address registers
  – efficient use of address registers and specialized address generation units

• Code compression
  – reduction of memory, important for cost sensitive applications
TMS320C25 - Instructions (1)

- Addition

  \[ \text{add} \rightarrow a \]

  \[ \text{add} \rightarrow a \quad \text{m (memory)} \]

- Subtraction

  \[ \text{spac} \rightarrow a \]

  \[ \text{spac} \rightarrow a \quad \text{p} \]
TMS320C25 - Instructions (2)

- Multiplication

\[
\begin{align*}
\text{mpy} & \rightarrow p \\
* & \rightarrow p \\
t & \rightarrow m \\
\text{mpyk} & \rightarrow p \\
* & \rightarrow const \\
t & \rightarrow \text{const}
\end{align*}
\]

- Data transfer

\[
\begin{align*}
lack & \rightarrow \text{const} \\
lac & \rightarrow a \\
pac & \rightarrow a \\
lac & \rightarrow m \\
lac & \rightarrow a \\
lac & \rightarrow m \\
sac & \rightarrow a \\
sac & \rightarrow m
\end{align*}
\]
\[ X := A \cdot B - (C+D) \cdot E \]
DAG – Example (2)

; left subtree
; first: cost 12

\[ \begin{align*}
\text{lt} & \rightarrow M_A \\
\text{mpy} & \rightarrow M_B \\
\text{pac} & \\
\text{sacl} & \rightarrow M_0 \\
\text{lac} & \rightarrow M_C \\
\text{add} & \rightarrow M_D \\
\text{sacl} & \rightarrow M_1 \\
\text{lt} & \rightarrow M_E \\
\text{mpy} & \rightarrow M_1 \\
\text{lac} & \rightarrow M_0 \\
\text{spac} & \\
\text{sacl} & \rightarrow M_X
\end{align*} \]
DAG – Example (3)

; right subtree
; first: cost 14

lac M_C
add M_D
sacl M_0
lt M_E
mpy M_0
pac
sacl M_1
lt M_A
mpy M_B
pac
lt M_1
mpyk 1
spac
sacl M_X
DAG - Example (4)

; optimal code
; cost 10

lac \rightarrow M_C
add \rightarrow M_D
sacl \rightarrow M_0
lt \rightarrow M_A
mpy \rightarrow M_B
pac
lt \rightarrow M_E
mpy \rightarrow M_0
spac
sacl \rightarrow M_X

previous method based on dynamic programming is no longer optimal!
Register Transfer Graph

- **Definition**: The register transfer graph for a processor architecture is a directed graph where each node represents a location in the data path at which data can be stored. An edge between two nodes $r_i$ and $r_j$ is labelled with the instructions that read operands from $r_i$ and write operands to $r_j$. 

![Diagram of a register transfer graph with nodes labeled m and a connected by an edge labeled lac]
RTG - Criterion

- **Definition:** The RTG criterion is satisfied if for all node triples \((r_1, r_2, r_3)\) of the RTG for which

  1. \(r_3\) has incoming edges from register nodes \(r_1\) and \(r_2\) with identical labelling, and

  2. there exists at least one cycle in the RTG including \(r_1\) and \(r_2\),

the following holds:

- Each cycle between \(r_1\) and \(r_2\) includes a memory node.
1 add: \( a = a + m \)
2 apac: \( a = a + p \)
3 spac: \( a = a - p \)
4 mpy: \( p = t \times m \)
5 mpyk: \( p = t \times \text{const} \)
6 lack: \( a = \text{const} \)
7 pac: \( a = p \)
8 sacl: \( m = a \)
9 lac: \( a = m \)
10 lt: \( t = m \)

RTG criterion satisfied!
• For processor architectures that satisfy the RTG criterion, optimal code can be generated in $O(n)$ time.
  – code generation in 2 phases
    ▪ optimal instruction selection and register binding (based on dynamic programming)
    ▪ optimal scheduling
  – $n$ is the number of DAG nodes
Address Generation Units (AGUs)

- TMS320C2x
- ADSP 210x
- Motorola 56K
AGU - Characteristics

- Set of address registers $AR$ for indirect addressing modes

- Set of modification registers $MR$ for updating the $AR$

- Modification operations in parallel to instruction execution
  - eg. post-modify: auto increment/decrement by
    - one address (+/- 1) or
    - the content of an $MR$
<table>
<thead>
<tr>
<th>operation</th>
<th>function</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR load</td>
<td>AR[ARP] = value</td>
<td>1</td>
</tr>
<tr>
<td>MR load</td>
<td>MR[MRP] = value</td>
<td>1</td>
</tr>
<tr>
<td>ARP load</td>
<td>ARP = value</td>
<td>0</td>
</tr>
<tr>
<td>MRP load</td>
<td>MRP = value</td>
<td>0</td>
</tr>
<tr>
<td>immediate modify</td>
<td>AR[ARP] += value</td>
<td>1</td>
</tr>
<tr>
<td>auto-increment</td>
<td>AR[ARP] ++</td>
<td>0</td>
</tr>
<tr>
<td>auto-decrement</td>
<td>AR[ARP] --</td>
<td>0</td>
</tr>
<tr>
<td>auto-modify</td>
<td>AR[ARP] += MR[MRP]</td>
<td>0</td>
</tr>
</tbody>
</table>
AGU – Problems

• Allocation of memory addresses and address modification registers
  – scalar variables:
    ▪ What is an optimal ordering of variables in memory, given 1 AR/n AR?
    ▪ How to efficiently use MR?
  – array variables:
    ▪ How to efficiently use AR/MR for loops? (important for DSPs)
Addressing of Scalar Variables

- access sequence: \texttt{b, d, a, c, d, a, c, b, a, d, a, c, d}
- 1 \texttt{AR} available

\begin{align*}
\text{LOAD AR, 1} \\
\text{AR } &+= 2 \\
\text{AR } &-= 3 \\
\text{AR } &+= 2 \\
\text{AR } &++ \\
\text{AR } &-= 3 \\
\text{AR } &+= 2 \\
\text{AR } &-- \\
\text{AR } &--
\end{align*}

\begin{align*}
\text{LOAD AR, 3} \\
\text{AR } &-- \\
\text{AR } &-- \\
\text{AR } &+= 2 \\
\text{AR } &-- \\
\text{AR } &-- \\
\text{AR } &+= 3 \\
\text{AR } &-= 2 \\
\text{AR } &++ \\
\text{AR } &-- \\
\text{AR } &+= 2
\end{align*}
Access Graph

- **Definition:** The access graph for a given set of variables and an access sequence consists of
  - nodes that model the variables,
  - edges that connect nodes if the corresponding variables are adjacent in the access sequence
  - edge weights that denote the number of transitions between the corresponding variables in the access sequence

An optimal address allocation is given by a Hamiltonian path in the access graph (path that visits each node exactly once) with maximal edge weight.

- NP-hard problem
Access Graph - Example (1)

access sequence: b, d, a, c, d, a, c, b, a, d, a, c, d
Access Graph - Example (2)

- additionally 1 MR available:

```
LOAD AR, 3
AR --
AR --
AR --
AR += 2
AR --
AR --
AR += 3
AR -= 2
AR ++
AR --
AR --
AR += 2
```

```
LOAD AR, 3
AR --
AR --
AR --
LOAD MR, 2
AR += MR
AR --
AR --
AR += 3
AR -= MR
AR ++
AR --
AR --
AR += MR
```
Addressing of Array Variables

• Typical assumptions for DSP code:
  – nested for loops with known number of iterations
  – each (nesting) level has a loop counter which is in/de-cremented by one
  – reference of an array element:

\[ A[\mathbf{f}_1(i_1)][\mathbf{f}_2(i_2)]\cdots[\mathbf{f}_n(i_n)] \]

indexing function: \( f_j(i_j) = \{c, i_j \pm c, c \pm i_j\} \)

\( c \in \mathbb{N}_0 \)

loop counter: \( i_j \)
Memory Layout

• $n$-dimensional array

$$A[m_1][m_2] \cdots [m_n] \quad m_i \in [0, N_i - 1]$$

$$adr(A[m_1][m_2] \cdots [m_n]) = base(A) + \sum_{j=1}^{n} m_j a_j$$

$$a_j = \begin{cases} 
\prod_{k=j+1}^{n} N_k & 1 \leq j < n \\
1 & j = n 
\end{cases}$$

• Example: array definition $A[10][4][5]$

$$adr(A[5][3][1]) = base(A) + 5 \cdot (4 \cdot 5) + 3 \cdot 5 + 1 \cdot 1 = base(A) + 116$$
Update Value

- **Simple loop:**
  
  ```
  FOR i=low TO high DO 
    ref 1
    ref 2
    ...
    ref k
  ENDFOR
  ```

- **Definition:** The update value $UV$ of an array reference is:

  $$
  UV(A[f_1(i)][f_2(i)]\cdots[f_n(i)]) = adr(A[f_1(i_1)]\cdots[f_n(i_1)]) -
  adr(A[f_1(i_0)]\cdots[f_n(i_0)])
  $$

  where $i_0$ is the index of an iteration and $i_1$ the index of the next iteration.
Update Value - Example

FOR i=2 TO 1024 DO
    (1) ref A[i+1]
    (2) ref A[i]
    (3) ref A[i+2]
    (4) ref A[i-1]
    (5) ref A[i+1]
    (5) ref A[i]
    (7) ref A[i-2]
ENDFOR

AR1 = adr(A[3])
FOR i=2 TO 1024 DO
    AR1 --
    AR1 += 2
    AR1 -= 3
    AR1 += 2
    AR1 --
    AR1 -= 2
    AR1 += 4
ENDFOR

references with identical UVs may share an address register:
Code Compression

• Target code is redundant and can be compressed
  – GP systems: decompression at program loading time
  – for embedded systems the reduction of program ROMs, RAMs is important

• Decompression in the cache

• External Pointer Macro (EPM) model
  – dictionary: contains frequently used code sequences (mini-subroutines)
  – skeleton: contains instructions and pointers to the dictionary
  – implementation in SW or HW
EPM Model in Hardware

[Diagram of EPM model in hardware]

- Control
- Dictionary Mode
- PUSH\(_0\)
- POP\(_0\)
- Counter
- Link Register Stack
- Program Counter
- MUX
- Program Bus
Compiler and Code Generation

- Compiler structure, intermediate code
- Code generation
- Code optimization
- Code generation for specialized processors
- Retargetable compiler
Retargetable Compiler

• Portable Compiler
  – developer retargetable
  – code generation by tree pattern matching

• Compiler-compiler
  – user retargetable (semi-automatic)
  – compiler is generated from a description of the target architecture (processor model)

• Machine independent compiler
  – automatically retargetable
  – compiler generates code for several processors / processor variants
  – for parametrizable architectures
Tree Pattern Matching

- Rules for transforming a syntax tree (DAG) are given as tree patterns

```
replacement ← pattern { action }
```

example:
```
reg i ← + { ADD Rj, Ri }

reg i

reg j
```

stepwise replacement by tree pattern matching until the tree contains only one node
Target Instructions (1)

(1) \( \text{reg } i \leftarrow \text{const } c \{ \text{MOV } \#c, \text{Ri} \} \)

(2) \( \text{reg } i \leftarrow \text{mem } a \{ \text{MOV } a, \text{Ri} \} \)

(3) \( \text{mem } \leftarrow \{ \text{MOV } \text{Ri}, \text{a} \} \)

(4) \( \text{mem } \leftarrow \{ \text{MOV } \text{Rj}, \ast \text{Ri} \} \)

(5) \( \text{reg } i \leftarrow \text{ind } \{ \text{MOV } c(\text{Rj}), \text{Ri} \} \)
Target Instructions (2)

(6) \( \text{reg } i \leftarrow \{ \text{ADD } c(\text{Rj}), \text{Ri} \} \)

\[ \begin{align*}
\text{reg } i & \quad \text{ind} \\
\text{const } c & \quad \text{reg } j
\end{align*} \]

(7) \( \text{reg } i \leftarrow + \{ \text{ADD } \text{Rj}, \text{Ri} \} \)

\[ \begin{align*}
\text{reg } i & \quad \text{reg } j
\end{align*} \]

(8) \( \text{reg } i \leftarrow + \{ \text{INC } \text{Ri} \} \)

\[ \begin{align*}
\text{reg } i & \quad \text{const } 1
\end{align*} \]
Tree Pattern Matching - Example (1)

\[ a[i] := b + 1 \]

(1) \{ MOV #a, R0 \}

Diagram:

```
:=
    +
  ind
+ mem b
  +
const 1
  +
ind
  +
const i
  +
reg SP
  +
const a
  +
reg SP
```
Tree Pattern Matching - Example (2)

\[
a[i] := b + 1
\]
Tree Pattern Matching - Example (3)

\[ a[i] := b + 1 \]
Tree Pattern Matching - Example (4)

\[ a[i] := b + 1 \]

\[
\begin{align*}
\text{:=} & \quad \text{ind} \\
+ & \quad \text{reg 0} \\
& \quad \text{mem b} \\
& \quad \text{const 1}
\end{align*}
\]

(2) \{ \text{MOV b, R1} \}
Tree Pattern Matching - Example (5)

\[
a[i] := b + 1
\]

\[
\begin{array}{c}
\text{ind} \\
\text{reg 0} \\
\text{reg 1} \\
\text{const 1}
\end{array}
\]

(8) \{ \text{INC R1} \}
Tree Pattern Matching - Example (6)

\[ a[i] := b + 1 \]

(4) \{ MOV R1, *R0 \}

MOV #a, R0
ADD SP, R0
ADD i(SP), R0
MOV b, R1
INC R1
MOV R1, *R0
Compiler Compiler:
Instruction Set Extraction

**instruction**  \( \text{xx011zz} \)

**operation**  \( \text{reg[zz] <- reg[xx] + acc} \)

**pattern**  
\[
\begin{align*}
\text{reg} & \quad + \\
\text{acc} & \quad \text{reg}
\end{align*}
\]
Processor Models

• Approaches for evaluating performance of processors without hardware implementation

• Behavioral models
  – describe the instruction set
  – simulation relatively fast (100-1000 times slower than target machine)
  – not very accurate (no pipelining effects)

• Structural models
  – describe the processor on the register transfer level
  – accurate
  – simulation rather slow

• Mixed models