A New Compilation Technique for SIMD Code Generation across Basic Block Boundaries

Hiroaki Tanaka, Yutaka Ota, Nobu Matsumoto
Center for Semiconductor Research and Development,
Semiconductor Company, Toshiba, corp.

Takuji Hieda, Yoshinori Takeuchi, Masaharu Imai
Graduate School of Information Science and Technology,
Osaka University

Jan. 19, 2010
Outline

- Background
  - SIMD Code Generation across Basic Block Boundaries
  - Experimental Results
  - Conclusion
Background

• **Spread of digital signal processing applications**
  – CODEC, speech/image recognition, etc.
  – Require higher computational power for microprocessors

• **SIMD instruction sets**
  – Provide both high performance and reasonable energy consumption
SIMD instructions

• Perform multiple operations on data in parallel
  – Take registers having several small data

• Difficulties in the utilization of SIMD instructions
  – Need to extract parallel operations from sequential programs
SIMD Code Generation Techniques

• **For loops** [Bik, et al. 2002]
  
  – Map same operations in different iterations into one instruction

  ```
  for (i=0;i<N;i++)
  C[i] = A[i] + B[i];
  ```

  ![Diagram showing SIMD operations]

• **For basic blocks** [larsen, et al. 2000]
  
  – Map same operations in a basic block into one instruction

  ```
  C[i] = A[i] + B[i]
  C[i+1] = A[i+1] + B[i+1]
  ```

  ![Diagram showing SIMD operations]
SIMD Code Generation with if-conversion

• For programs with control flow
  – Remove conditional statements using “if-conversion”
    
    ```
    if(x==0) {
      c[i] = a[i]+b[i];
      c[i+1] = a[i+1]+b[i+1];
    } else {
      c[i] = a[i]-b[i];
      c[i+1] = a[i+1]-b[i+1];
    }
    ```

    ```
    t0 = a[i]+b[i];
    t1 = a[i+1]+b[i+1];
    t2 = a[i]-b[i];
    t3 = a[i+1]-b[i+1];
    ```

    ```
    c[i] = (x==0) ? t0 : t2;
    c[i+1] = (x==0) ? t1 : t3;
    ```

Limitations:

– If-conversion may decrease performance
– Not applicable when “if-else” statements cannot be removed by if-conversion
Challenge: SIMD Code Generation with Control Flow

\begin{align*}
a0 &= P0[0]; \\
a1 &= P0[1]; \\
b0 &= P1[0]; \\
b1 &= P1[1]; \\
if (\text{mode}) & \{ \\
    a0 &= a0 + b0; \\
    a1 &= a1 + b1; \\
\} & \text{else} & \{ \\
    a0 &= a0 - b0; \\
    a1 &= a1 - b1; \\
\}
\end{align*}

\begin{align*}
P2[0] &= a0; \\
P2[1] &= a1;
\end{align*}

Key point: the way to keep data dependency between basic blocks
Outline

• Background

SIMD Code Generation across Basic Block Boundaries

• Experimental Results

• Conclusion
SIMD Code Generation across Basic Block Boundaries

• Generates SIMD code without modifying control flow
  – Deal with data dependency between basic blocks

• Basic concepts
  – Extension of basic block vectorization
  – Vectorization between basic blocks
  – Packing cost
Basic concepts

\[ a = P[0] \]
\[ b = P[1] \]

\[ Y = X \]

\[ Y = Z \]

Extends existing bb vectorization

Vectorize between bbs.

Use packing cost
Proposed Code Generation Flow

1. Input program
2. SSA transformation
3. For each basic block:
   a. Making packed vars.
   b. Vectorization within a bb.
   c. Insertion of assignments for packed vars.
4. Removal of redundant copies
5. Output program

Additional steps:
- Packing cost
- Extension of basic block vectorization
- Vectorization between basic blocks
SSA transformation

• Remove implicit dependency due to variables
  – Use existing SSA transformation technique

```c
a0 = P0[0];
a1 = P0[1];
b0 = P1[0];
b1 = P1[1];
if (mode) {
a0 = a0 + b0;
a1 = a1 + b1;
} else {
a0 = a0 - b0;
a1 = a1 - b1;
}
P2[0] = a0;
P2[1] = a1;
```

```c
a00 = P0[0];
a10 = P0[1];
b00 = P1[0];
b10 = P1[1];
if (mode) {
a01 = a00 + b00;
a11 = a10 + b10;
} else {
a02 = a00 - b00;
a12 = a10 - b10;
}
a03 = \varnothing(a01, a02);
a13 = \varnothing(a11, a12);
P2[0] = a03;
P2[1] = a13;
```
Vectorization within a Basic Block

- Make packed variables based on packing cost, then, vectorize each basic block

```plaintext
a00=P0[0];
a10=P0[1];
b00=P1[0];
b10=P1[1];
if(mode) {
    a01=a00+b00;
    a11=a10+b10;
} else {
    a02=a00-b00;
    a12=a10-b10;
}
a03=\otimes(a01, a02);
a13=\otimes(a11, a12);
P2[0]=a03;
P2[1]=a13;
```

A0 : (a00,a10)
B0 : (b00,b10)

A0=P0[0:1];
B0=P1[0:1];
if(mode) {
    A2=A1+B1
} else {
    A4=A3+B2
}
P2[0:1]=A5

(a00,a10) : 1
(a00,b00) : 2
(a00,b10) : 2
(a10,b00) : 2
(a10,b10) : 2
(b00,b10) : 1
Insertion of assignments for packed vars.

- Look up packed vars having same contents
- Then, insert an assignment statement

\[
\begin{align*}
A0 &= P0[0:1]; \\
A1 &= A0; \\
B0 &= P1[0:1]; \\
B1 &= B0; \\
\text{if}(\text{mode}) \{ \\
A2 &= A1 + B1; \\
A3 &= A0; \\
B2 &= B0; \\
\text{if}(\text{mode}) \{ \\
A4 &= A3 + B2; \\
A5 &= A2; \\
\} \\
\} \\
P2[0:1] &= A5; \\
A5 &= A3 - B2; \\
A5 &= A4; \\
\}
\end{align*}
\]
Removal of Redundant Copies

- A lot of redundant statements may be inserted

-> use traditional copy removal technique

```c
A0=P0[0:1];
B0=P1[0:1];
A1=A0;
B1=B0;
A3=A0;
B2=B0;
if(mode) {
    A2=A1+B1;
    A5=A2;
} else {
    A4=A3-B2;
    A5=A4;
}
P2[0:1]=A5;
```
Outline

• Background
• SIMD Code Generation across Basic Block Boundaries

Experimental Results

• Conclusion
Compiler Implementation

• **Target processor**
  – Toshiba MeP with IVC2 embedded processor core
    • 3 issue VLIW architecture
    • 8 byte/4 halfword/2 word SIMD instruction set

• **Compiler**
  – C compiler for MeP with IVC2 was used as a base compiler
  – Proposed technique was implemented in the base compiler
Average Performance Evaluation

- **Randomly generated programs**
  - Composed of:
    - 5 “if” statements
    - 2 SIMD operations in each basic block
    - 0-16 repacking of SIMD data
    - 4 elements in a SIMD datum
  - Use 10 programs for each 0,1,2,4,8,16 repacking

- **Evaluation method**
  - Each program was executed 32 times
  - Compared total execution cycle count over 10 programs with different repacking count
Result

Repacking count

- 16
- 8
- 4
- 2
- 1
- 0

Total execution cycle count

no SIMD
across bb SIMD

Jan. 19, 2010
ASP-DAC 2010
Case Study: H.264 decoder inter prediction

- **Reference software developed for MeP**
  - before MeP specific optimization
  - Inter prediction:
    - Consumes about 16% of execution cycles in entire decoding process
    - Contains 16 if-statements

- **Compared different compilation**
  - “no SIMD”
  - “loop and bb SIMD”
  - “across bb SIMD” (proposed)
  - “manual SIMD”
Comparison of Execution Cycle Count

![Bar Chart]

- No SIMD
- Loop and bb SIMD
- Across bb SIMD (proposed)
- Manual SIMD

# of execution cycle count

[K cycle.]

67% reduction
Breakdown of Static Instruction Counts

[Diagram showing instruction counts across bb SIMD (proposed) and manual SIMD.]

- Scalar inst.:
- Copy/pack inst.:
- SIMD inst.:

100 inst.
Outline

• Background
• SIMD Code Generation across Basic Block Boundaries
• Experimental Results

Conclusion
Conclusion

• New SIMD vectorization technique across basic block boundaries was proposed

• Performance of inter-prediction was improved
  – Showed a 67% reduction in execution cycles

• Future work
  – Enhance vectorization to reduce copy/pack instructions