Machine-Independent Compiler Optimizations for the UofT DSP Architecture

Sanjay M. Pujare and Corinna G. Lee
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada

Outline

DSP-Style Programming

Machine-Independent Optimizations

→ C-Style Programming

Performance Results

Summary and Conclusions

Using Hardware Do-Loop Instruction (A Processor-Specific Optimization)

```
    mov.d  #1,d10                    do     #10,LEND
LSTART: /* loop body */              /* loop body */
          :                               :
          :                               :
LTEST:                          LEND:
    mov.d  #1,d11                         :
    add.d  d10,d11,d10
    mov.d  #10,d12
    slt.d  d10,d12,d13
    bnz.d  d13,LSTART
```

✓ faster execution because fewer instructions

✗ can only be used when

- number of iterations known in advance at run-time
- no references to loop index variable

Need to Use DSP-style Programming for Performance

C-style

```
for (i = 0; i < N; i++) {
    b[i] = a[i];
}
```

✗ cannot translate into hardware do-loop instruction because:

- loop-index variable **`i`** is used inside the loop

DSP-style

```
pa = a; pb = b;
for (i = 0; i < N; i++) {
    *pb++ = *pa++;
}
```

✓ can translate into hardware do-loop instruction because:

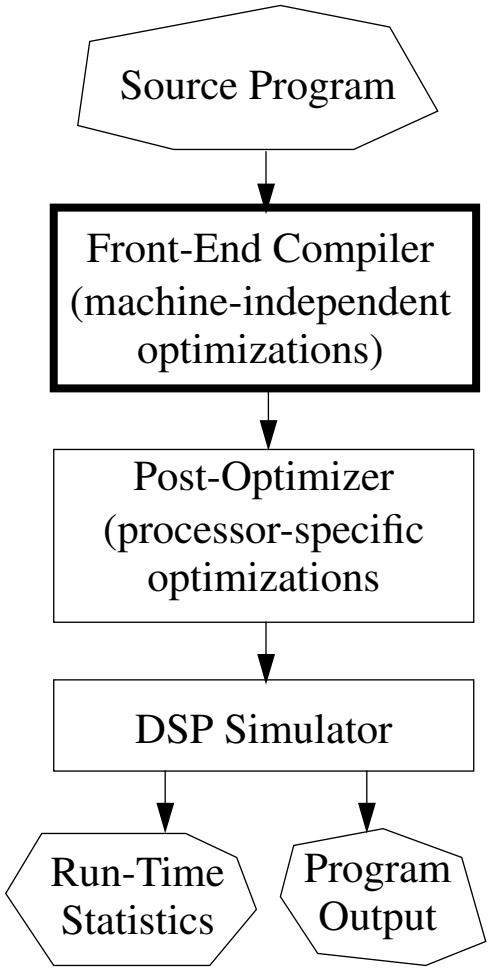
- no reference to loop-index variable by introducing extra variables
- use of pointers to access array elements

✗ less readable

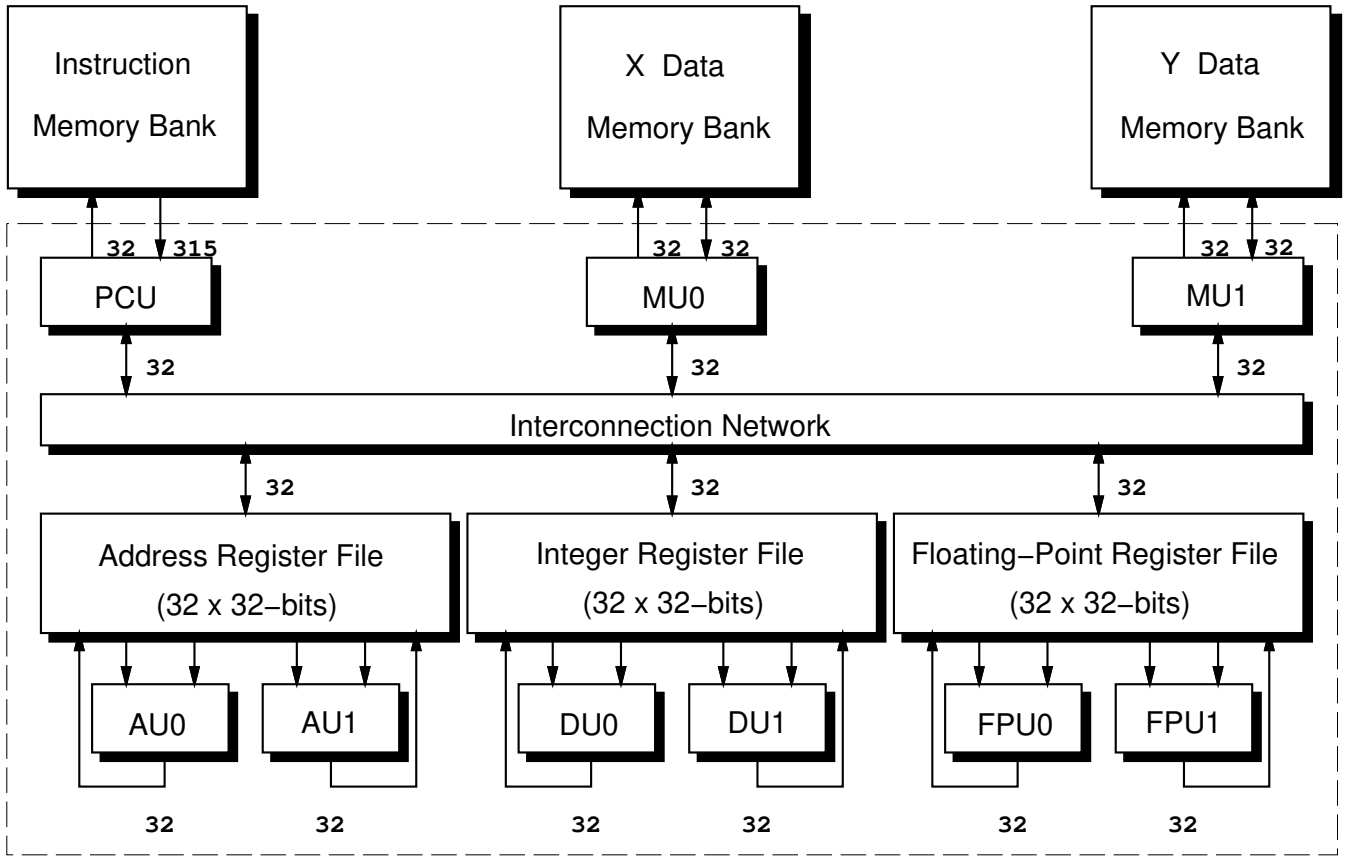
✗ low-level programming

How much performance could be lost when using C-style programming?

UofT DSP System



UofT DSP Architecture

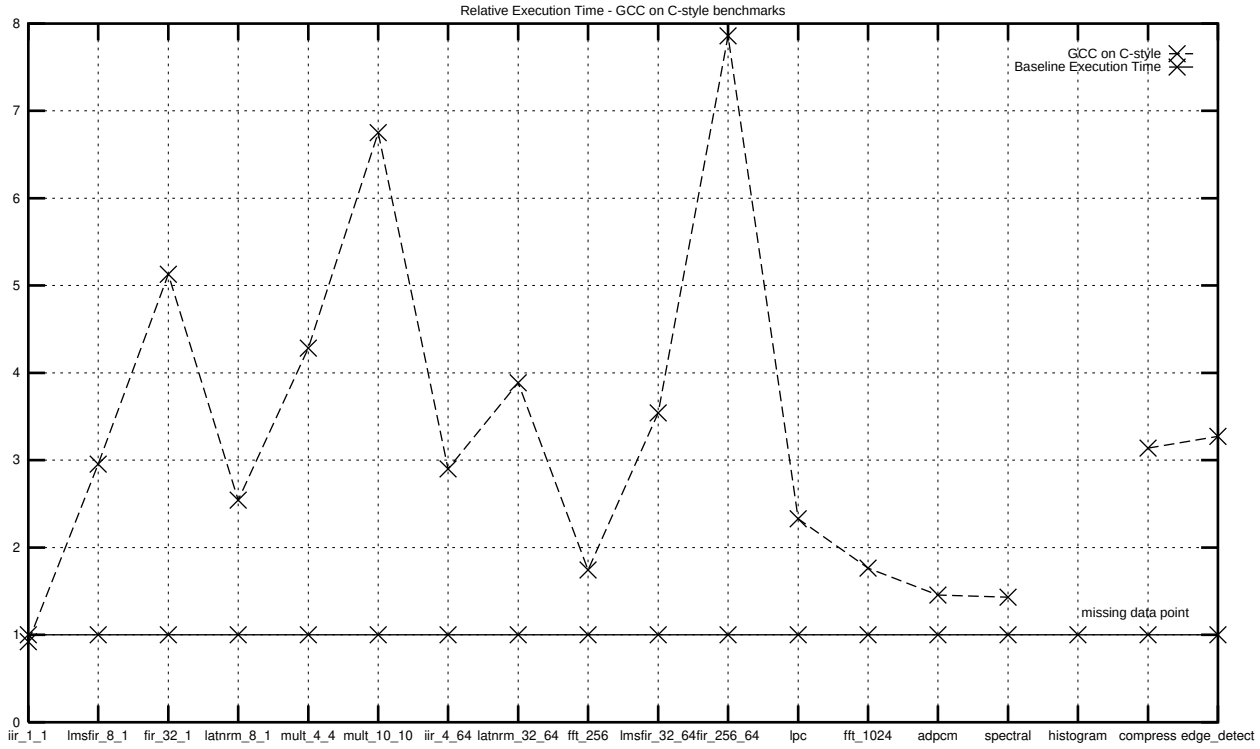


Performance of C-Style & DSP-Style Programming

C-style programming

DSP-style programming

Performance Relative to DSP-Style Programming



Outline

DSP-Style Programming

Machine-Independent Optimizations

→ **C-Style Programming**

Performance Results

Summary and Conclusions

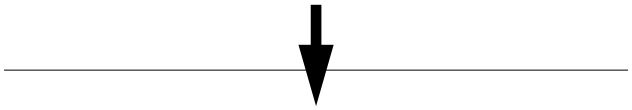
Machine-Independent Scalar Optimizations to Eliminate DSP-Style Programming

- goal: use C-style programming without sacrificing performance (i.e., enable the use of processor-specific optimizations)
- machine-independent scalar optimizations include
  - loop-invariant code motion (LICM)
  - induction variable creation (IVC) and strength-reduction
  - common sub-expression elimination
- replace GCC-based front-end compiler with SUIF compiler
- implemented optimizations in compiler phase called INDVAR

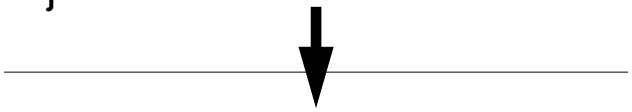
Loop-Invariant Code Motion (LICM)

- move constant expressions out of loop body
- enables induction variable creation in nested loops

```
for (i = 0; i < N; i++) {
  for (j=0; j < M; j++) {
    a[i][j] = 5 * j;
  }
}
```



```
for (i = 0; i < N; i++) {
  for (j=0; j < M; j++) {
    *(a+M*i+j) = 5*j;
  }
}
```

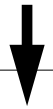


```
for (i = 0; i < N; i++) {
  pa = a+M*i;
  for (j=0; j < M; j++) {
    *(pa+j) = 5*j;
  }
}
```

Induction Variable Creation (IVC)

- eliminates uses of loop-index variables
- affine expressions involving loop-index variables are replaced with induction variables (special case of strength reduction)

```
for (i = 0; i < N; i++) {
  pa = a+M*i;
  for (j=0; j < M; j++) {
    *(pa+j) = 5*j;
  }
```

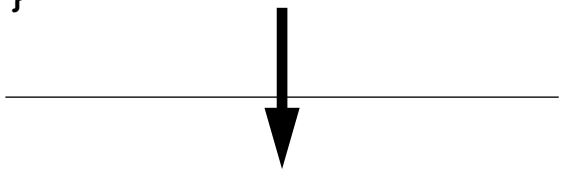


```
ivar_pa = a;
for (i = 0; i < N; i++) {
  ivar1_j = pa = ivar_pa;
  ivar2_j = 0;
  for (j=0; j < M; j++) {
    *(ivar1_j) = ivar2_j;
    ivar1_j ++;
    ivar2_j += 5;
  }
  ivar_pa += M;
}
```

Common Sub-Expression Elimination

- implemented as part of LICM and IVC
- same expressions replaced by same induction variable

```
for (i = 0; i < N; i++) {
  ... = i * 4;
        :
  ... = i * 4;
}
```



```
for (i = 0; i < N; i++) {
  ... = ivar_i;
        :
  ... = ivar_i;
}
```

Outline

DSP-Style Programming

Machine-Independent Optimizations

→ C-Style Programming

Performance Results

Summary and Conclusions

Benchmark Suite

18 programs:

- 12 basic filters of different sizes
- 6 algorithms for speech compression and image processing

several versions of each program

- —— DSP-style programming with software pipelining (SWP)
- —— C-style programming
- —— C-style programming with software pipelining

each program compiled with two front-end compilers

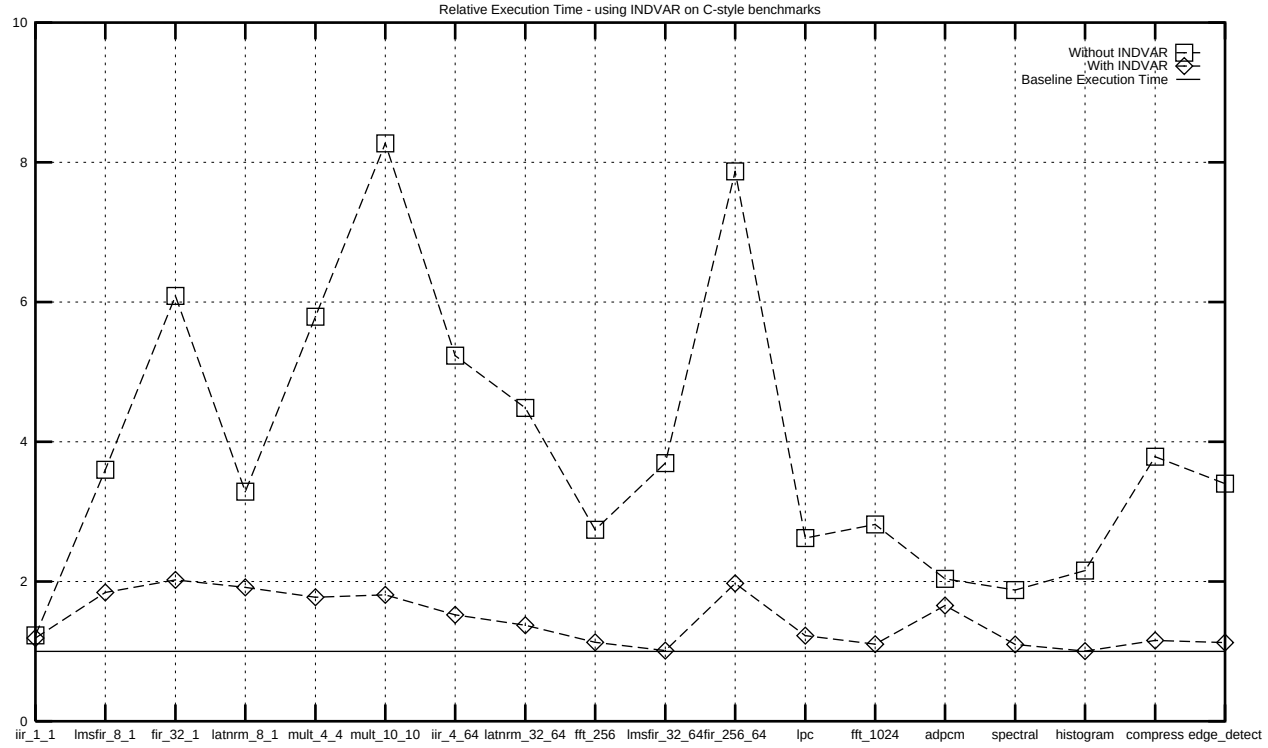
- × GCC-based
- □ SUIF-based without INDVAR optimization
- ◇ SUIF-based with INDVAR optimization

Performance Impact of INDVAR Optimizations

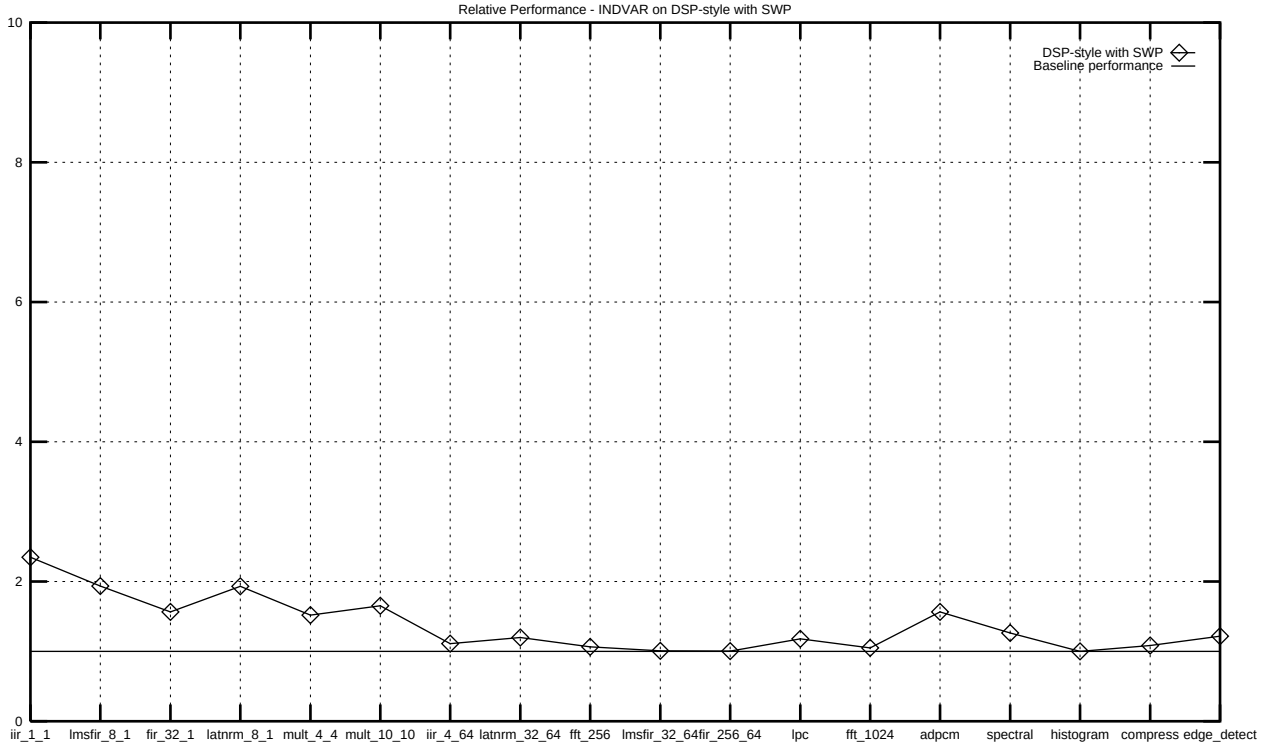
C-style without INDVAR

C-style with INDVAR

Performance Relative to DSP-Style Programming Using GCC



DSP-style with INDVAR & SWP using SUIF



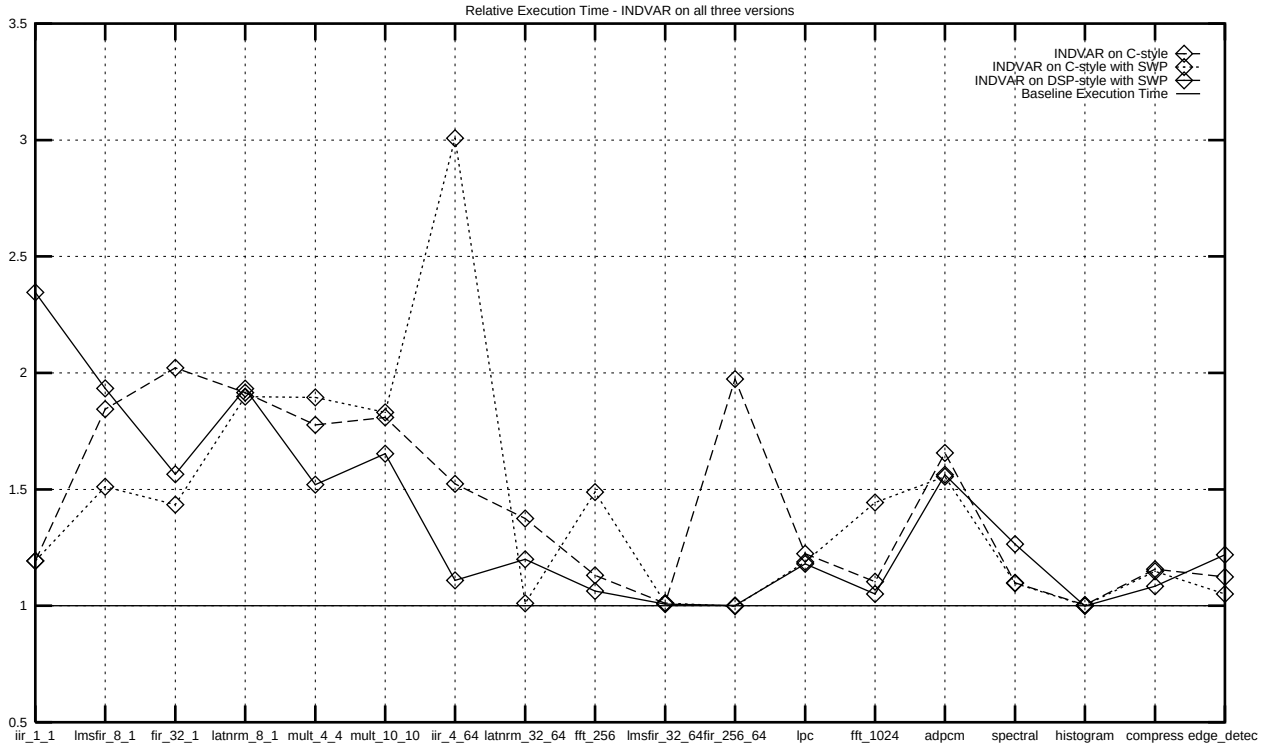
SWP Programming and INDVAR Optimizations

C-style with INDVAR

C-style with INDVAR & SWP

DSP-style with INDVAR & SWP using SUIF

Performance Relative to DSP-Style Programming Using GCC



Conclusions

- to utilize the full power of high-level languages without sacrificing performance, it is necessary to implement both machine-independent optimizations and processor-specific optimizations
- example: loop-invariant code motion (LICM) and induction variable creation (IVC) are machine-independent optimizations that enable the use of the hardware DO instruction, a processor-specific optimization
- to obtain consistent high performance, compiler should implement all types of optimizations