Compilers Wrap-up

Combination of three lectures
15-213
“The course that gives CMU its Zip!”

Code Optimization I:
Machine Independent Optimizations
Sept. 26, 2002

Topics
- Machine-Independent Optimizations
  - Code motion
  - Reduction in strength
  - Common subexpression sharing
- Tuning
  - Identifying performance bottlenecks
Great Reality #4

There’s more to performance than asymptotic complexity

Constant factors matter too!

- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels:
  - algorithm, data representations, procedures, and loops

Must understand system to optimize performance

- How programs are compiled and executed
- How to measure program performance and identify bottlenecks
- How to improve performance without destroying code modularity and generality
Optimizing Compilers

Provide efficient mapping of program to machine
- register allocation
- code selection and ordering
- eliminating minor inefficiencies

Don’t (usually) improve asymptotic efficiency
- up to programmer to select best overall algorithm
- big-O savings are (often) more important than constant factors
  - but constant factors also matter

Have difficulty overcoming “optimization blockers”
- potential memory aliasing
- potential procedure side-effects
Limitations of Optimizing Compilers

Operate Under Fundamental Constraint
- Must not cause any change in program behavior under any possible condition
- Often prevents it from making optimizations when would only affect behavior under pathological conditions.

Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
- e.g., data ranges may be more limited than variable types suggest

Most analysis is performed only within procedures
- whole-program analysis is too expensive in most cases

Most analysis is based only on static information
- compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative
Machine-Independent Optimizations

- Optimizations you should do regardless of processor / compiler

Code Motion

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop

```c
for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```
Compiler-Generated Code Motion

- Most compilers do a good job with array code + simple loop structures

Code Generated by GCC

```c
for (i = 0; i < n; i++) {
    int ni = n*i;
    int *p = a+ni;
    for (j = 0; j < n; j++)
        *p++ = b[j];
}
```

```assembly
imull %ebx,%eax  # i*n
movl 8(%ebp),%edi  # a
leal (%edi,%eax,4),%edx  # p = a+i*n (scaled by 4)

.L40:
    movl 12(%ebp),%edi  # b
    movl (%edi,%ecx,4),%eax  # b+j (scaled by 4)
    movl %eax,(%edx)  # *p = b[j]
    addl $4,%edx  # p++ (scaled by 4)
    incl %ecx  # j++
    jl .L40  # loop if j<n
```
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \quad \rightarrow \quad x \ll 4 \]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + j] = b[j];
```

```c
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```
Make Use of Registers

- Reading and writing registers much faster than reading/writing memory

Limitation

- Compiler not always able to determine whether variable can be held in register
- Possibility of Aliasing
- See example later
Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

```c
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n
1 multiplication: i*n
Time Scales

Absolute Time
- Typically use nanoseconds
  - $10^{-9}$ seconds
- Time scale of computer instructions

Clock Cycles
- Most computers controlled by high frequency clock signal
- Typical Range
  - 100 MHz
    - $10^8$ cycles per second
    - Clock period = 10ns
  - 2 GHz
    - $2 \times 10^9$ cycles per second
    - Clock period = 0.5ns
- Fish machines: 550 MHz (1.8 ns clock period)
Cycles Per Element

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- T = CPE*n + Overhead

![Graph showing Cycles Per Element](image-url)
Code Motion Example #2

Procedure to Convert String to Lower Case

```c
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Extracted from 213 lab submissions, Fall, 1998
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
void lower(char *s)
{
    int i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
    done:
}

- **strlen** executed every iteration
- **strlen** linear in length of string
  - Must scan string until finds '\0'
- Overall performance is quadratic
Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```c
void lower(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance

![Graph showing CPU seconds vs string length for lower case conversion performance. The graph compares two methods: lower1 and lower2. The x-axis represents string length in bytes, ranging from 256 to 262144, and the y-axis represents CPU seconds, ranging from 0.000001 to 1000.]
Optimization Blocker: Procedure Calls

*Why couldn’t the compiler move `vec_len` or `strlen` out of the inner loop?*

- Procedure may have side effects
  - Alters global state each time called
- Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure lower could interact with `strlen`

*Why doesn’t compiler look at code for `vec_len` or `strlen`?*

- Linker may overload with different version
  - Unless declared static
- Interprocedural optimization is not used extensively due to cost

**Warning:**

- Compiler treats procedure call as a black box
- Weak optimizations in and around them
Reduction in Strength

```c
void combine3(vec_ptr v, int *dest) {
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest += data[i];
    }
}
```

Optimization

- Avoid procedure call to retrieve each vector element
  - Get pointer to start of array before loop
  - Within loop just do pointer reference
  - Not as clean in terms of data abstraction

- CPE: 6.00 (Compiled -O2)
  - Procedure calls are expensive!
  - Bounds checking is expensive
Eliminate Unneeded Memory Refs

```c
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}
```

**Optimization**

- Don’t need to store in destination until end
- Local variable `sum` held in register
- Avoids 1 memory read, 1 memory write per cycle
- CPE: 2.00 (Compiled -O2)
  - Memory references are expensive!
Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location

Example

- $v: [3, 2, 17]$
- $\text{combine3}(v, \text{get_vec_start}(v)+2) \rightarrow ?$
- $\text{combine4}(v, \text{get_vec_start}(v)+2) \rightarrow ?$

Observations

- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing
Machine-Independent Opt. Summary

**Code Motion**
- Compilers are good at this for simple loop/array structures
- Don’t do well in presence of procedure calls and memory aliasing

**Reduction in Strength**
- Shift, add instead of multiply or divide
  - Compilers are (generally) good at this
  - Exact trade-offs machine-dependent
- Keep data in registers rather than memory
  - Compilers are not good at this, since concerned with aliasing

**Share Common Subexpressions**
- Compilers have limited algebraic reasoning capabilities
Important Tools

Measurement

- Accurately compute time taken by code
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
- Profile procedure calling frequencies
  - Unix tool gprof

Observation

- Generating assembly code
  - Lets you see what optimizations compiler can make
  - Understand capabilities/limitations of particular compiler
Code Profiling Example

Task
- Count word frequencies in text document
- Produce sorted list of words from most frequent to least

Steps
- Convert strings to lowercase
- Apply hash function
- Read words and insert into hash table
  - Mostly list operations
  - Maintain counter for each unique word
- Sort results

Data Set
- Collected works of Shakespeare
- 946,596 total words, 26,596 unique
- Initial implementation: 9.2 seconds

Shakespeare’s most frequent words

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>29,801</td>
<td>the</td>
</tr>
<tr>
<td>27,529</td>
<td>and</td>
</tr>
<tr>
<td>21,029</td>
<td>I</td>
</tr>
<tr>
<td>20,957</td>
<td>to</td>
</tr>
<tr>
<td>18,514</td>
<td>of</td>
</tr>
<tr>
<td>15,370</td>
<td>a</td>
</tr>
<tr>
<td>14010</td>
<td>you</td>
</tr>
<tr>
<td>12,936</td>
<td>my</td>
</tr>
<tr>
<td>11,722</td>
<td>in</td>
</tr>
<tr>
<td>11,519</td>
<td>that</td>
</tr>
</tbody>
</table>
Code Profiling

Augment Executable Program with Timing Functions

- Computes (approximate) amount of time spent in each function
- Time computation method
  - Periodically (~ every 10ms) interrupt program
  - Determine what function is currently executing
  - Increment its timer by interval (e.g., 10ms)
- Also maintains counter for each function indicating number of times called

Using

gcc -O2 -pg prog. -o prog
./prog
  - Executes in normal fashion, but also generates file gmon.out

gprof prog
  - Generates profile information based on gmon.out
## Profiling Results

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>cumulative seconds</th>
<th>time</th>
<th>time seconds</th>
<th>calls</th>
<th>self seconds</th>
<th>ms/call</th>
<th>total</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.60</td>
<td>8.21</td>
<td>8.21</td>
<td>8.21</td>
<td>1</td>
<td>946596</td>
<td>8210.00</td>
<td>8210.00</td>
<td>sort_words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.80</td>
<td>8.76</td>
<td>0.55</td>
<td>946596</td>
<td>9.21</td>
<td>0.45</td>
<td>0.00</td>
<td></td>
<td>lower1</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>4.75</td>
<td>9.21</td>
<td>0.45</td>
<td>946596</td>
<td>9.33</td>
<td>0.12</td>
<td>0.00</td>
<td></td>
<td>find_ele_rec</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>1.27</td>
<td>9.33</td>
<td>0.12</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>h_add</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

## Call Statistics
- Number of calls and cumulative time for each function

## Performance Limiter
- Using inefficient sorting algorithm
- Single call uses 87% of CPU time
Code Optimizations

- First step: Use more efficient sorting function
- Library function `qsort`
Further Optimizations

- Iter first: Use iterative function to insert elements into linked list
  - Causes code to slow down
- Iter last: Iterative function, places new entry at end of list
  - Tend to place most common words at front of list
- Big table: Increase number of hash buckets
- Better hash: Use more sophisticated hash function
- Linear lower: Move `strlen` out of loop
Profiling Observations

**Benefits**
- Helps identify performance bottlenecks
- Especially useful when have complex system with many components

**Limitations**
- Only shows performance for data tested
- E.g., linear lower did not show big gain, since words are short
  - Quadratic inefficiency could remain lurking in code
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds
Profiling, Instrumentation, and Profile Based Optimization

Robert Cohn
Robert.Cohn@compaq.com

Mark T. Vandevoorde

COMPAQ
Introduction

Understanding the dynamic interaction between programs and processors

– What do programs do?
– How do processors perform?
– How can we make it faster?
What to do?

Build tools!

– Profiling
– Instrumentation
– Profile based optimization
The Big Picture

Sampling -> Profiling

Instrumentation

Profiling ->

Analysis

Profile Based Optimization

Modeling
Instrumentation

• User level view
• Executable editing
Code Instrumentation

- Application appears unchanged
- Data collected as a side effect of execution
Instrumentation Example

• Add extra code

```c
if (b > c) {
    t = 1;
    bb[0]++;
} else {
    bb[1]++;
    t = 1;
    b = 3;
}
```
Instrumentation Uses

- Profiles
- Model new hardware
  - What will this new branch predictor do?
  - What is the miss rate of this new cache?
- Optimization opportunities
  - find unnecessary loads and stores
  - find divides by 1
What Tool Does Instrumentation?

• Compiler
  – Compiler inserts extra operations
  – Requires recompile, access to source code

• Executable editor
  – Post-link tool inserts instrumentation code
  – No rebuild, source code not required
  – More difficult to relate back to source
Executable editors

- Input: executable, output: executable
- Instrument, optimize, translate
- Executable = image = binary = shared library = shared object = dynamically linked library (DLL)
- Executable editor, executable optimizer, binary rewriter, binary translator, post link optimizer
Intermediate Representation

• Similar to compiler
  – except unstructured, untyped data
  – 1 to 1 mapping for IR and machine instructions

• Base representation should be compact
  – fit in physical memory
    • initial/final phases do multiple passes

• Representations built/thrown away for procedures
Other tools that change addresses

• Linker
  – combine separately compiled objects
  – adjust addresses based on assigned load address
  – unit is section of object (data, text)

• Loader
  – Load address != link address for DLL
  – unit is entire image

• Use relocations
Optimization for Performance: Compiler Options

Gregory Bauer
NCSA
gbauer@ncsa.uiuc.edu
Outline

• Let the compiler do the work
• But first …
• Optimization and Vectorization
  – Maximize instruction level parallelism
  – Reduce shared-resource contention
• Auto-Parallelization
  – Exploit application level parallelism
• Correctness

Images from Intel
Let the compiler do the work

• Help the compiler understand your code
• Look at what the compiler says it can and cannot do
• Make changes when appropriate

But first …

• **Why two compiler suites: Intel and GNU?**
  – Keep choices to a minimum for simplicity
  – Uniformity between NCSA Intel platforms (IA32, IA64, Intel 64)
  – Note current GNU version 3.4 (not 4) on Abe.

• **Which should you use?**
  – One that works for your application.

• **What about other compiler suites?**
  – PGI and PathScale support Intel 64 but are not installed.
Intel References

Intel Manuals
“Intel® 64 and IA-32 Architectures Optimization Reference Manual”
Compilers
[www.intel.com/software/products/compilers/docs/flin/main_for](http://www.intel.com/software/products/compilers/docs/flin/main_for)

Intel Books
Vector Programming - [www.intel.com/intelpress/sum_vmmx.htm](http://www.intel.com/intelpress/sum_vmmx.htm)

Intel Blackbelt Guide
Intel Samples

- Try codelets in `${INTEL_HOME}/samples` with the different reporters

<table>
<thead>
<tr>
<th>Compiler Area</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic optimizations</td>
<td>./samples/optimize/</td>
<td>Illustrates how to use the automatic compiler options to quickly change the performance of an application.</td>
</tr>
<tr>
<td>Profile-guide Optimization (PGO)</td>
<td>./samples/pgo_samples/</td>
<td>Illustrates profiling an application, generating PGO compiler reports, and using the code-coverage and test-prioritization tools.</td>
</tr>
<tr>
<td>Interprocedural Optimization (IPO)</td>
<td>./samples/ipo_samples/</td>
<td>Illustrates using multi-file IPO compilation and generating IPO compiler reports.</td>
</tr>
<tr>
<td>OpenMP* support</td>
<td>./samples/openmp_samples/</td>
<td>Illustrates using OpenMP* pragmas to create multi-threaded applications.</td>
</tr>
<tr>
<td>Vectorization</td>
<td>./samples/vec_samples/</td>
<td>Illustrates loops that will and will not vectorize on specific architectures. These samples also illustrate using the vectorization reporting features.</td>
</tr>
</tbody>
</table>
Optimizations

Focus on Intel compiler

• Start with: -O1, -O2, -O3
  -O1 Limit code size (no loop unrolling, …)
  -O2 Intra-file interprocedural optimizations (inlining …), loop unrolling, …
  -O3 High level optimizations (loop reordering, …), array padding

Some defaults:
- ftz at O1 and higher (-no-ftz to disable)
– vectorization at O2 and higher (-vec- to disable)
Optimizations

What about -fast?

-fast

“enable -xT -O3 -ipo -no-prec-div -static”

-static can conflict with bss/zero initialized data and loading of libraries:

use -shared-intel

-static inhibits use of tools that use pre-loading from working
Optimizations

Make things clearer to compiler (when valid to do so):

- restrict "keyword to disambiguate pointers"
- ansi "ANSI standard conformance"
- ansi-alias" program adheres to aliasability rules"
- fno-alias "no aliasing in program"
- fno-fnalias "no aliasing within functions"

Use directives: loop count, unroll
Optimizations

Improve inlining with -ip or –ipo
- Removes cost (cycles and memory) of procedure call
- Build user libraries with Intel xiar for -ipo info
- -inline-level=0|1|2 to control
- Additional controls as well
- -Winline for information, also see opt-report

Images from Intel
Optimizations

Enable aggressive FP optimizations (which may not be safe)
- `fp-model fast=2` "aggressive optimizations on floating-point data"
- `fno-exceptions` "disables exception handling"
- `fno-math-errno` "no test of errno from math calls"
- `complex-limited-range` ""
Optimizations

prefetch

– Intel 64 core arch supports automatic hardware prefetching (-prefetch is for IA-64)
– Helps hide memory access time 😊
– prefetch/noprefetch directives for IA-64
– prefetch intrinsics can help
  • mm_prefetch
  • control cache destination, temporal-ness
  • can inhibit vectorization 😞
Optimization Report

Use optimization report to see what the compiler is doing (and what it is NOT doing)

- `opt-report[n] n=0(none),1(min),2(med),3(max)`
- `opt-report-file<file>`
- `opt-report-phase<phase>` or all
- `opt-report-routine<routine>`
Matrix multiplication

Matrix multiplication is an useful example
Best to use BLAS GEMM routine of course (MKL, GOTO, …)

```plaintext
38       do i = 1, n
39          do j = 1, n
40             do k = 1, n
41                 a(i,j) = a(i,j) + b(i,k)*c(k,j)
42             end do
43          end do
44       end do
```
Optimization Report

- Sample from matrix-matrix multiplication

```
<mxm.f;38:38;hlo_scalar_replacement;in MAIN__;0>
# of Array Refs Scalar Replaced in MAIN__ at line 38=36
# of Array Refs Scalar Replaced in MAIN__ at line 38=3

<mxm.f;38:38;hlo_linear_trans;MAIN__;0>
LOOP INTERCHANGE in loops at line: 38 39 40
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )

<mxm.f;38:38;hlo_unroll;MAIN__;0>
Loop at line 38 blocked by 113

<mxm.f;40:40;hlo_unroll;MAIN__;0>
Loop at line 40 blocked by 113

<mxm.f;39:39;hlo_unroll;MAIN__;0>
Loop at line 39 blocked by 113
Loop at line 39 unrolled and jammed by 4

<mxm.f;40:40;hlo_unroll;MAIN__;0>
Loop at line 40 unrolled and jammed by 4
```
Optimization Report

• F90 IPO sample

```<ipo_sample_main.f90;37:45;IPO INLINING;MAIN__;0>
INLINING REPORT: (MAIN__) [1/4=25.0%]

  -> for_write_seq_lis(EXTERN)
  -> INLINE: mysum_(5) (isz = 25) (sz = 32 (13+19))
      -> INLINE: add3_(6) (isz = 2) (sz = 7 (3+4))
      -> init_(0) (isz = 13) (sz = 20 (11+9))
  -> for_set_reentrancy(EXTERN)
</ipo_sample_main.f90;37:45>
```

```<ipo_sample_main.f90;37:45;IPO FORWARD SUBSTITUTION;MAIN__;0>
SUBSTITUTION: TOTAL(12): REPLACE(6) REMOVE(5) ADDR-TAKEN-OFF(1)
</ipo_sample_main.f90;37:45>
```

```<ipo_sample_sum.f90;-1:-1;IPO DEAD STATIC FUNCTION ELIMINATION;mysum_;0>
DEAD STATIC FUNCTION ELIMINATION:
  (mysum_)
  Routine is dead static
</ipo_sample_sum.f90>-1:-1>
```

```<ipo_sample_init.f90;40:48;IPO INLINING;init_;0>
INLINING REPORT: (init_) [4/4=100.0%]
</ipo_sample_init.f90;40:48>
```
Vectorization

- Vectorization with SSE2 at -O2
  - All Intel 64 processors support SSE2
- Additional SSE3/SSSE3: -xT
- Exploit parallelism in processor using MMX/SSE1/SSE2/SSE3/SSSE3 instructions
- Loop vectorization depends on operand data types and operators.

Images from Intel
Vectorization

• **Need**
  – Single loop entry/exit, trip-countable
  – Aligned data, limited data dependencies
  – +, -, *, /, negation, $\sqrt{}$, MAX, MIN, and math functions such as SIN, COS …

• **Alignment, Stride, Data dependencies**
  -align (C users use with caution), -pad
  – intrinsics: __declspec(align(16)), _mm_malloc/free
  – directives: vector {aligned | unaligned | always}, ivdep

• **Avoid**
  – equivalence in Fortran
  – function calls in loop
Vectorization Report

- Enabled at -O2 and higher
- `-vec-report[n]` control diagnostic information:
  - 0 no diagnostic information
  - 1 indicate vectorized loops (DEFAULT)
  - 2 indicate vectorized/non-vectorized loops
  - 3 indicate vectorized/non-vectorized loops and prohibiting data dependence information
  - 4 indicate non-vectorized loops
  - 5 indicate non-vectorized loops and prohibiting data dependence information
Vectorization Report

• Again, matrix multiplication

```fortran
38       do i = 1, n
39          do j = 1, n
40             do k = 1, n
41                a(i,j) = a(i,j) + b(i,k)*c(k,j)
42             end do
43          end do
44       end do
```

```plaintext
(stdout)
mxm.f(38): (col. 7) remark: PERMUTED LOOP WAS VECTORIZED.
mxm.f(39): (col. 10) remark: loop was not vectorized: not inner loop.
mxm.f(40): (col. 13) remark: loop was not vectorized: not inner loop.

(optimization_report_file)
mxm.f(38:7-38:7):VEC:MAIN__: PERMUTED LOOP WAS VECTORIZED
mxm.f(39:10-39:10):VEC:MAIN__: loop was not vectorized: not inner loop
mxm.f(40:13-40:13):VEC:MAIN__: loop was not vectorized: not inner loop
```
Vectorization Report

• An example with data dependencies

```fortran
33 subroutine flow_dependence(y)
34   implicit none
35   integer :: i
36   real, dimension(10), intent(inout) :: y
37   ! Flow dependency y accesses.
38   do i=2,10
39     y(i) = y(i-1) + 1
40   end do
41 end subroutine flow_dependence
```

HPO Vectorizer Report (flow_dependence_)

novec.f90(38): (col. 3) remark: loop was not vectorized: existence of vector dependence.
novec.f90(39): (col. 5) remark: vector dependence:
   proven FLOW dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_: loop was not vectorized: existence of vector dependence
Source code changes

• Hand loop unrolling and loop distribution can inhibit compiler optimizations.
• The optimization and vectorization reports can tell you what is preventing optimization or vectorization.
• With this information, making changes to the source may make optimization or vectorization possible and beneficial.
Profile Guided Optimization

- `prof-dir<d>` directory for profiling output
- `prof-file<f>` file profiling summary
- `prof-gen[x]` instrument program
- `prof-use` use profiling information

Images from Intel
Profile Guided Optimization

• Can help with trip-count and alias issues.

• Limited by what input used for application: e.g. may exercise code differently than production input

• Used in quite a few SPEC benchmark submissions
Profile Guided Optimization

• PGO report from -opt-report

```
<pgotools_sample.f90;-1:-1;PGO;MAIN__;0>
  DYN-VAL: pgotools_sample.f90 MAIN__

<pgotools_sample.f90;-1:-1;PGO;main_.main_add_;0>
  DYN-VAL: pgotools_sample.f90 main_.main_add_

<pgotools_sample.f90;-1:-1;PGO;addermod_.;0>
  NO-DYN:  pgotools_sample.f90 addermod_.

<pgotools_sample.f90;-1:-1;PGO;delegate_;0>
  DYN-VAL: pgotools_sample.f90 delegate_

<pgotools_sample.f90;-1:-1;PGO;addermod_mp_mod_add_;0>
  DYN-VAL: pgotools_sample.f90 addermod_mp_mod_add_

<;-1:-1;PGO;;0>
```

4 FUNCTIONS HAD VALID DYNAMIC PROFILES
1 FUNCTIONS HAD NO DYNAMIC PROFILES

• Degree of coverage
Auto-Parallelization

• Similar to -openmp
• Let the compiler try
  -parallel
  -par-threshold[n] - set loop count threshold
• Directive: parallel, noparallel

• Keep in mind MPI
Auto-Parallelization Report

-par-report\{0|1|2|3\} control the auto-parallelizer diagnostic level

```
<stdout>
mxm.f(160): (col. 7) remark: LOOP WAS AUTO-PARALLELIZED.
mxm.f(39): (col. 10) remark: PERMUTED LOOP WAS AUTO-PARALLELIZED.

<optimization_report_file>
HPO THREADIZER REPORT (MAIN__) LOG OPENED ON Mon Jul  2 21:52:21 2007
<mxm.f;-1:-1;hpo_threadization;MAIN__;0>
HPO Threadizer Report (MAIN__)
mxm.f(39:10-39:10):PAR:MAIN__: PERMUTED LOOP WAS AUTO-PARALLELIZED
mxm.f(40:13-40:13):PAR:MAIN__: loop was not parallelized: existence of parallel
dependence
mxm.f(38:7-38:7):PAR:MAIN__: loop was not parallelized: insufficient inner loop
```
Correctness

- Optimizations often affect results of computation.
- HLO: try lower optimization level
- Vectorization: disable with -vec-
- FP: control assumptions but still try high level of optimization
Correctness

- **-fp_model strict**
  value-safe optimizations for floating-point calculations, floating-point exception semantics.

- **-fp-speculation strict**
  disable speculation on floating-point operations.

- **-fltconsistency | -mieee-fp**
  improved floating-point consistency

- **-prec-div | -prec-sqrt**
  fully precise division/sqrt implementations
May help with multi-core

-opt-streaming-stores always

Default is “auto”. “stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.”
Interesting switches

• -sox “save the compiler options and version number in the executable“
Let the compiler do the work

• Help the compiler understand your code
• Look at what the compiler says it can and cannot do
• Make changes when appropriate
• Rinse and Repeat