Optimizing Loops

David Levinthal
Senior Software Engineer
Intel Corp

Sept 2005

Agenda

- Optimizing loops: What matters?
- What is a Short tripcount loop?
- SMG2000*
- Versioning loops

Comments are Specific to Loop Dominated Codes
Identification of Occurrences Mostly Relies on
Visual Inspection of Assembler + Performance Counters

Application Characterization: 2 Types

- Data Structure Driven
  - Execution is instruction retirement driven
    - Mostly FP op retirement
  - Memory access is Bandwidth dominated
    - Regular sequential array element access
    - very effective HW prefetch (IA-32 and Intel® 64 Architecture)
    - And SW prefetch (Itanium® 2 Processors, 9.0 compiler and later)
  - HPC: loop dominated apps

Application Characterization: 2 Types

- Data Value Driven
  - Execution is branch dominated
  - Memory access is Latency dominated
    - Data value, pointer chasing for linked lists
    - Prefetching is difficult
  - Database, Search Engines etc
- NOT DISCUSSED HERE

What Matters in loop

1. The Trip Count
2. The Trip Count
3. The TRIP COUNT!
4. Variations in the tripcount
5. And some other things

   BUT..what you do about them depends on

   THE TRIP COUNT

And of course there are virtually no tools to assist you in determining this..other than printf

(you can use PIN..

or prof_gen and then generate an asm listing and get the basic block counts..)

NOTE: Disambiguation is an exception to this rule..

Why is the Trip Count so Overwhelmingly Important

- Prefetch generation is based off the inner loop.
  - The assumption is that the inner loop has a very long tripcount
  - The prefetch distance (in iterations) is then 500/cycles_per_iteration
  - So a one line do loop with a trip count of 100 is toast

There is no other reasonable default for the compiler

- Software Pipelining is effective ONLY if the Prologue and Epilogue are insignificant
  - So a loop with 5 stages had better have a tripcount >> 10
- Remainder loops from unrolling are really hard to optimize well..so the remainder better be insignificant

Managing Tripcount

- Get the longest loop you have, moved to be the inner loop
  - Loop interchange
  - Completely unroll inner loops
    - pragma loop count/pragma unroll may help
  - Data organization interchange..make loop index unit stride of array access
    - Leading dimension: first index for fortran, last for c/C++
    - This makes the prefetching work
    - Uses ALL the bandwidth
  - DO NOT DECREASE THIS DIMENSION WITH MPI PARTITIONING!!!!!!!

The Order of Optimization

- Get everything prefetched
  - Everything else is irrelevant if you wait 300 cycles every 16 variables..(or worse)
- Disambiguate the pointers
  - This kills SWP and vectorization
  - This can even be an issue in F77!!!
    - Once the code has been translated to Intermediate Language the identity of the source language (and its rules) are gone
    - Note: two references to the same array with different symbolic indexes are a true ambiguation
- Everything else:
  - Make sure minimum instruction count is being retired in inner loops (this comprises a lot of tricks)
  - Check latency is being hidden
  - Minimize Bandwidth
  - Check all asm versions of loops with significant cpu_cycles
  - Etc etc..we'll get to this

The Big Four, The Biggest Bang for the Buck

- CPU_CYCLES (SAV = 1,000,000)
- BACK_END_BUBBLE.ALL (SAV = 1,000,000)
- DEAR_LATENCY_GT_64 (SAV = 1000)
- BUS_MEMORY.ALL.SELF (SAV = 10,000)
  - Disable Calibration
- Identifies
  - Low hanging fruit with back_end_bubble.all
  - Hotspots (cpu_cycles / back_end_bubble.all)
  - Long latency data access/non prefetched data/thread sharing problems
  - Bandwidth limited loops

Why Those SAV Values?

- Source and asm views in VTune™ Performance Analyzer display sample counts
  - ratios of sample counts should be meaningful!
- Back_end_bubble.all/Cpu_Cycles = fraction of stalls
  - Sav values are equal so they cancel
- Dear_latency_gt_64/cpu_cycles ~ fraction of stalled cycles due to long latency loads

Dear * 1000 (SAV_dear)*300 (L3 miss latency)*3(~dear sampling)/cpu_cycles*1000,000 (SAV_cpu) = (900,000/1000,000)*dear/ cpu_cycles

~ dear/ cpu_cycles ~ stalled cycle fraction

BW limit ~ 5.5 bus cycles/bus transaction =>

5.5 < (cpu_cycles*1000,000*0.2Ghz/1.6 Ghz) /(Bus_memory*10,000) =100/8 *(cpu/bus)

44/100 < cpu/bus ~1/2

Visual Inspection of Assembler

- Single most powerful tool
  - Other than printf(tripcount) of course!!! ☺
- Compiler generates many versions of loops
  - For various address alignments
  - Disambiguation
  - Tripcounts
- Use VTune™ Performance Analyzer to figure out which one is actually executed
  - CPU sensitive assembly editor

Prefetching

- Nothing else matters as much
- Each L3 cache miss results in ~300 cycles
  - 20 cycles over 16 iterations if every element in an array is used
  - 300 cycles/element/iteration in the worst case!
- It is difficult for anything else to cause this large a problem

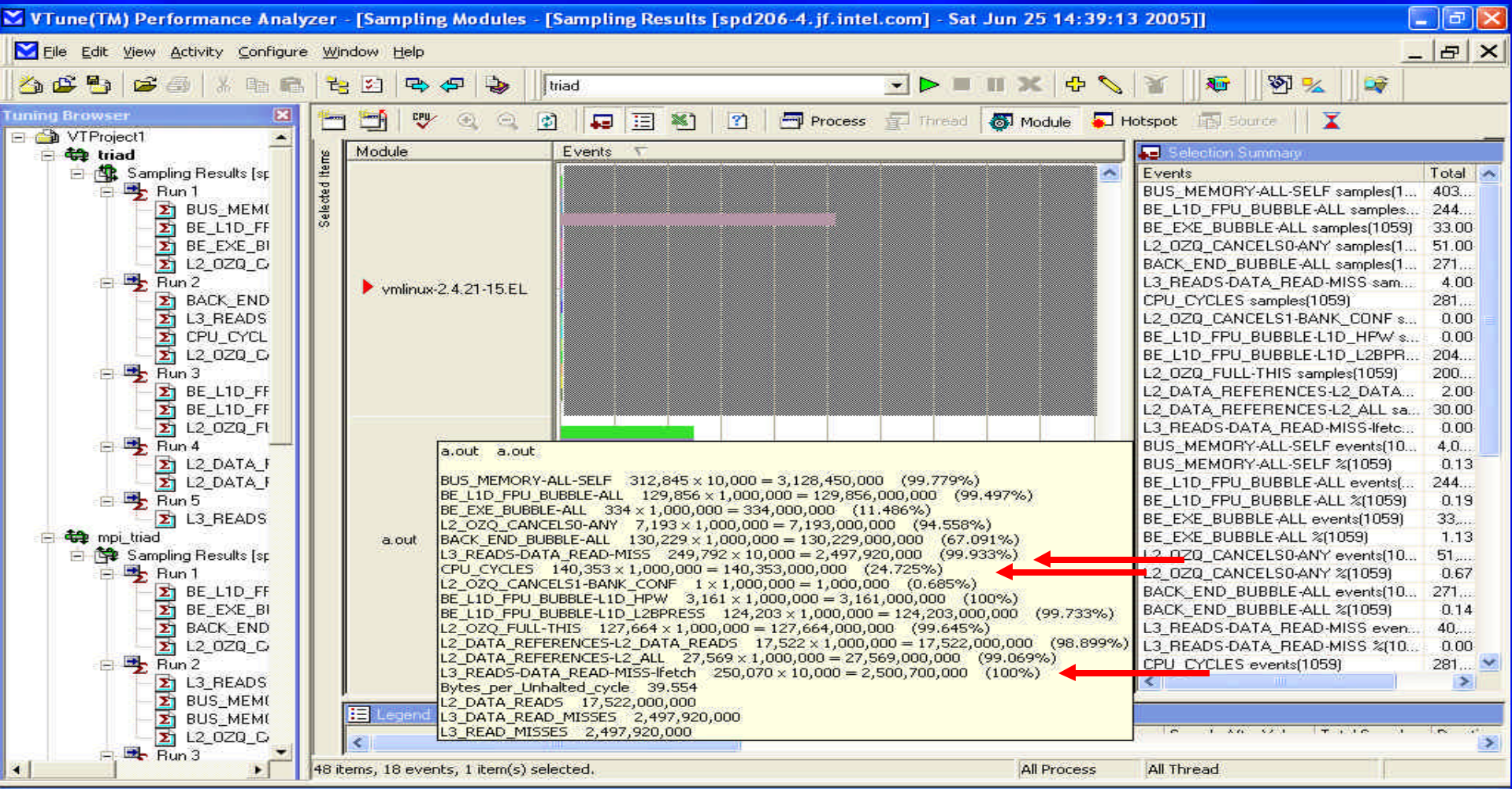
Identifying a Prefetch Problem

- Collect L3_reads.data_reads.miss with and without an opcode tag on lfetch
  - The difference is the size of your problem
  - SAV (L3_reads) = SAV(CPU)/100
- Collect DEAR_latency_gt_64
  - SAV(DEAR) = SAV(CPU)/1000
  - Fraction of stalled cycles due to long latency loads ~
    dear_latency_gt_64*300*3(sampling fraction)/cpu_cycles
    using samples: ~dear(samp)/cpu(samp)

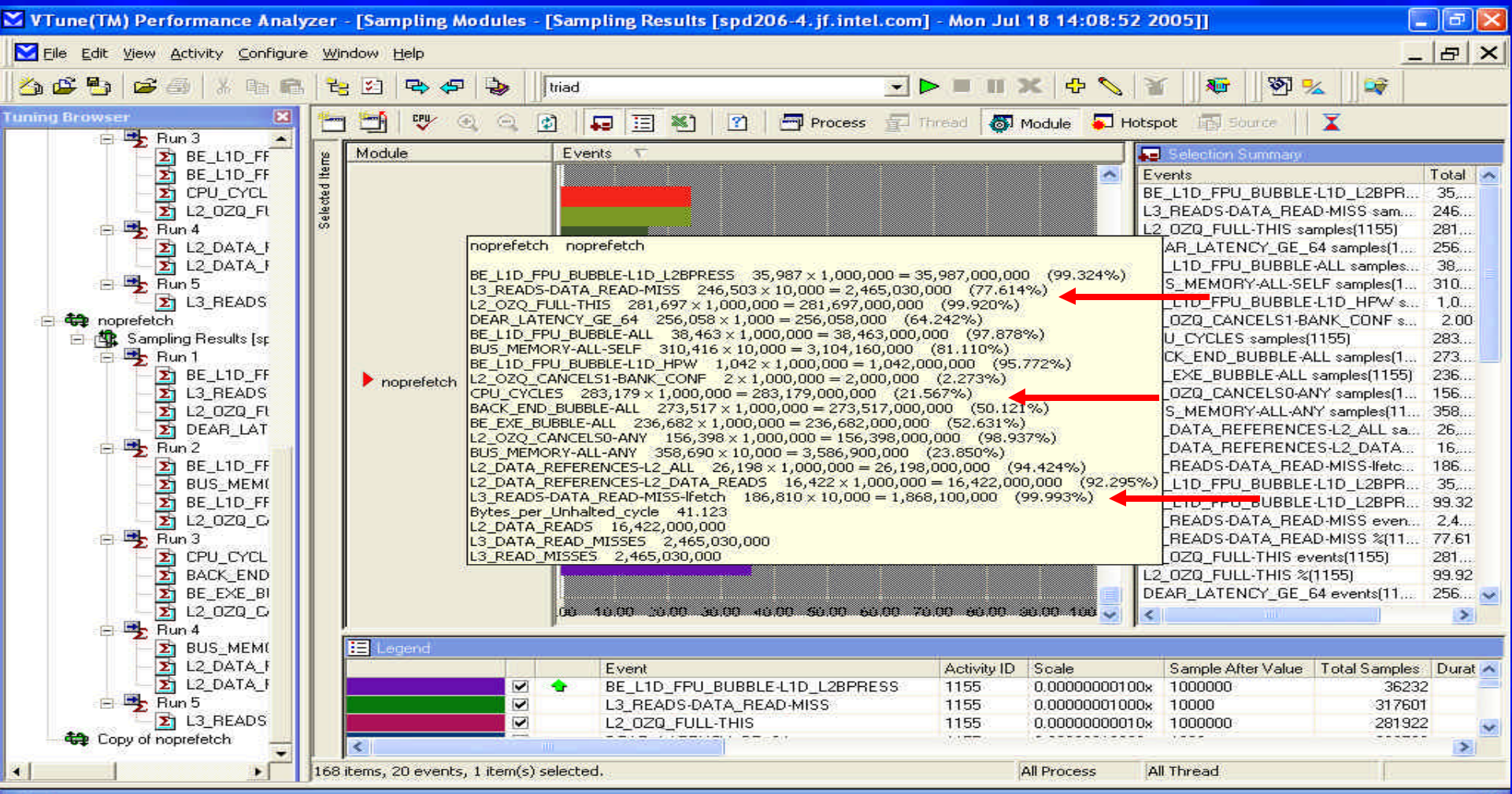
DEAR is exact (no skid) it identifies unprefetched load variables (use source line # in asm view to assist: multiple source lines with one array/line does not help!)

Then use lfetch intrinsic (__lfetch or mm_prefetch)

Bandwidth limited App Prefetching everything A(i) = B(i)+C(i)*D(i)



Bandwidth limited App Missing One Prefetch (pragma noprefetch c)



Default Compiler Prefetch Generation

- Compiler generates lfetch instructions driven by loop index of the innermost loop effecting data reference
  - Prefetch distance (iterations) ~ 500/cycles_per_iteration (asm loop)
  - Is available in compiler opt_report
- This works for long tripcount loops
  - Tripcount > prefetch distance
- Works for dense array access in multi dimensional loop, with unit stride data access pattern
  - Prefetches beyond inner loop tripcount, automatically prefetches for next iteration of outer loop

Using Lfetch Intrinsics

- Simple inner loops prefetch some number of iterations ahead
  - ~500/cycles_per_iteration
  - in the asm view.. count ;;
- If the tripcount is not >> prefetch distance..this fails
  - May result in additional, unused bandwidth
- LFETCH is a BW limited instruction!
  - 128 bytes/slot!!! Up to 512 bytes/cycle
- Redundant Lfetches are suppressed by OzQ
  - Dropped/cancelled secondary miss
  - Occupy memory slots

Mechanics of Lfetch Usage

- Two basic issues
  - Driving prefetches from outer loops
    - Inner loop is too short and can't be unrolled out of existence
    - First inner loop (of many) is BW limited
  - Indirect Addressing aka gather/scatter
    - One level gather usually done by compiler
    - Multilevel gather or compiler failure requires hand written prefetch
    - Tripcount is too short to support 2 level gather
- Prefetching Structures may require multiple lfetch
  - Structure size > 128 bytes
  - Structure poorly aligned to cachelines
    - Structure spans 2 cachelines
    - Prefetch leading and trailing edge of structure

Mechanics of Lfetch Usage with Outer Loops

- Inner loop has very short tripcount, but a lot of code
  - It cannot be unrolled away
- Resolution:
  - Create a prefetch value from the index of the outer loop
    - Outer_loop_index+ 1? 2?
    - You won't need much..there are a lot of instructions
  - Prefetch all the data being used for each inner iteration but using the forward outer_loop index
    - Gathers may result in a fair bit of code

Versioning Loops

- Different tripcounts and strides may have different optimal encodings
  - An inner loop with a tripcount that is always either 3 or 4 has 2 distinct unroll options
- Predicate Hoisting
- Address alignments may or may not allow runtime disambiguation
- Address alignments may allow or preclude load pairs

Generation of multiple loop encodings
Selected by runtime test

Versioning on Inner Tripcount

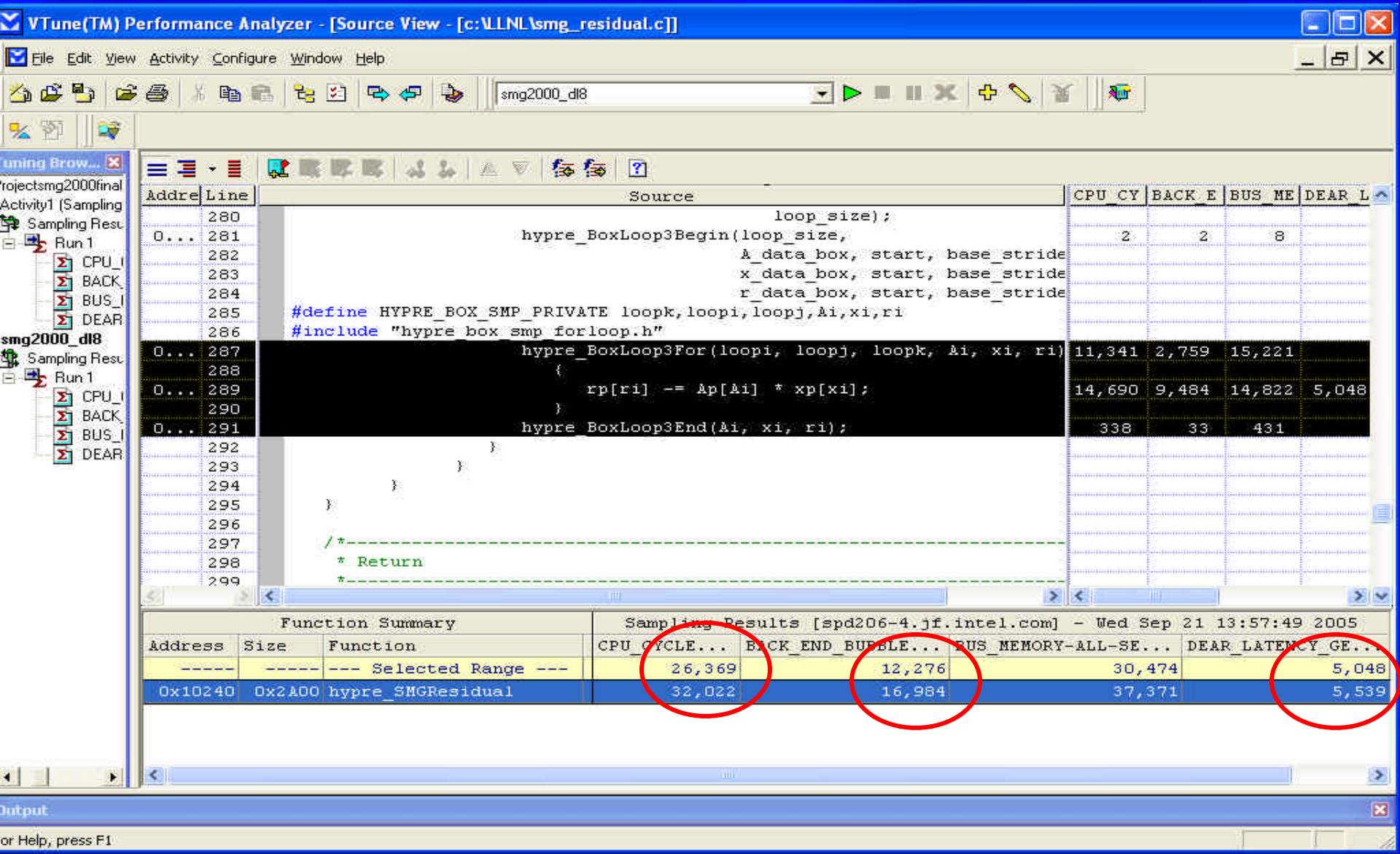
```
DO j=1, jmax
  DO k=1, kmax

    IF( phase .EQ. 3) THEN
      DO i=1, imax
        sum = ( density(i,k  ,j,1)+density(i,k  ,j,2)+density(i,k-1,j,3)+  &
 &                density(i,k-1,j,1)+density(i,k-1,j,2)+density(i,k  ,j,3) )  <---unrolled by 3
        csum(i,k,j) = 1./(1.+0.5*sum)   <---unroll allows SWP of divide
      ENDDO
    ELSE

      DO i=1, imax
        sum = 0.
        DO ispe=1,phase
          sum = sum + density(i,k,j,ispe) + density(i-1,k,j,ispe) <-short inner loops are lousy
        ENDDO
        csum(i,k,j) = 1./(1.+0.5*sum) <-divide outside inner loop
      ENDDO
    ENDIF

  ENDDO
ENDDO
```

PGO Will Do This Automatically

Versioning on Stride

- Variable stride creates havoc for the compiler
  - Makes strength reduction go out the window
  - Drives prefetch generation crazy
  - Makes unrolling very messy
  - In principle blocks simd generation
- Sources of variable stride
  - Explicit use of variables to increment array indicies
  - Dope vectors for F90 assumed shape arrays
- Reality is that these “variable” strides are really always 1
  - IA-32/Intel® 64 Architecture compiler versions on stride to allow simd generation

SMG2000*



How do we optimize a 1 Line Loop?

- In this case
- Expand the macros with the preprocessor
- Copy the expanded macro code for this one section
- Replace the expanded macro code for the original block which had the macros
  - Leave all other macros alone!
- Rewrite the code

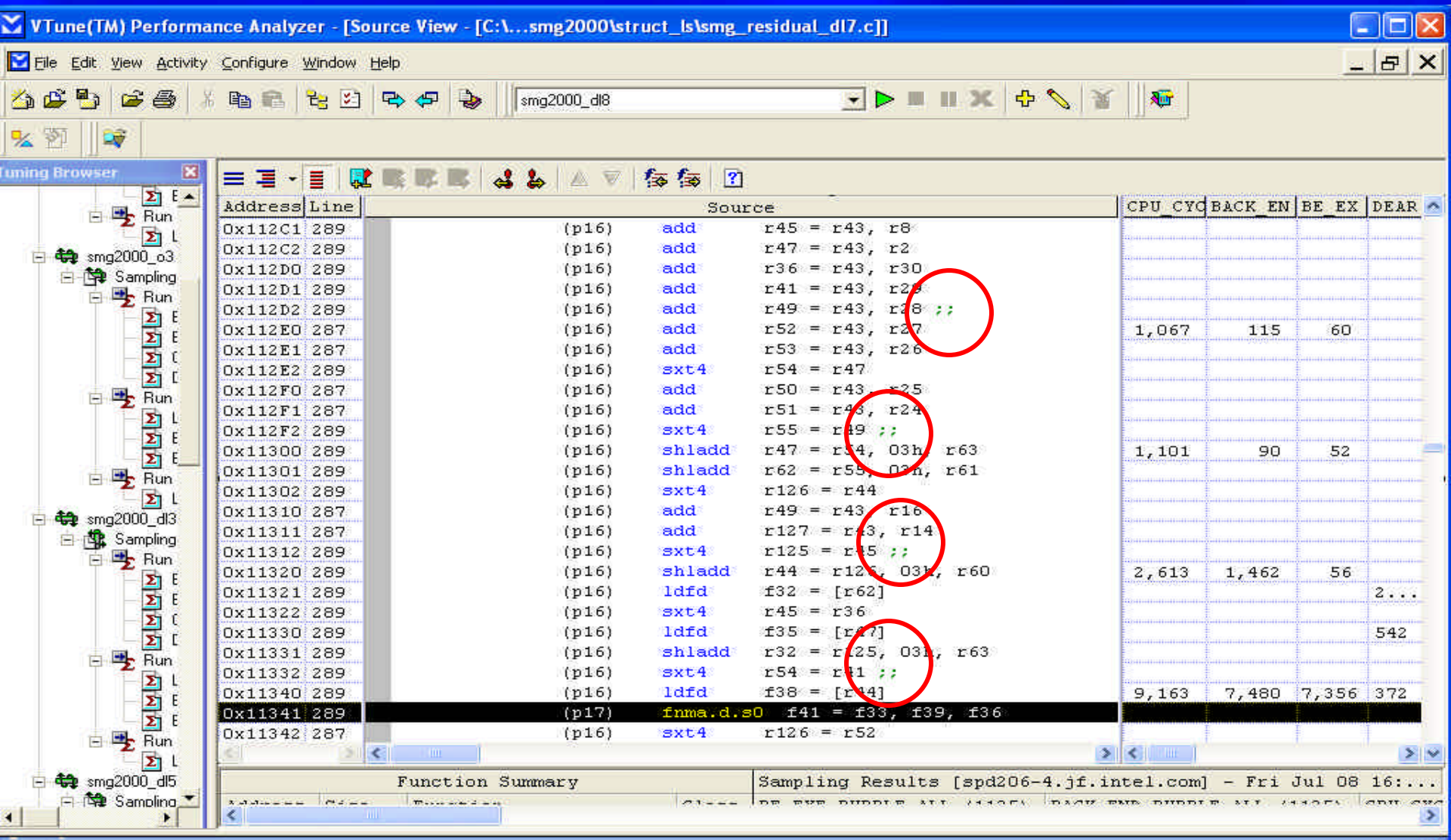
SMG2000*: Macros expanded

```
for (loopk = 0; loopk < hypre__nz; loopk++)
    {                                                    //7
    for (loopj = 0; loopj < hypre__ny; loopj++)
            {                                            //8
            for (loopi = 0; loopi < hypre__nx; loopi++)
                    {                                    //9
                    rp[ri] -= Ap[Ai] * xp[xi];
                    Ai += hypre__sx1;
                    xi += hypre__sx2;
                    ri += hypre__sx3;
                    }                                    //8
            Ai += hypre__sy1 - hypre__nx*hypre__sx1;
            xi += hypre__sy2 - hypre__nx*hypre__sx2;
            ri += hypre__sy3 - hypre__nx*hypre__sx3;
            }                                            //7
    Ai += hypre__sz1 - hypre__ny*hypre__sy1;
    xi += hypre__sz2 - hypre__ny*hypre__sy2;
    ri += hypre__sz3 - hypre__ny*hypre__sy3;
    }                                                    //6
```

SMG2000* Issues

- Inner loop tripcount is short compared to prefetch distance (hypre_nx = 120)
  - 1 fma/iteration -> 2 iterations/cycle -> prefetch “distance” ~ 600 iterations >> 120
- Variable inner stride could mean 1 cacheline/iteration
  - All 3 arrays must be prefetched every iteration
- Variable outer stride means noncontiguous memory access
- Assembler analysis reveals 11 cycles/ 2 iterations!
  - Instead of 1 cycle/2 iterations

SMG2000* Visual Inspection of Assembler



11 cycles per 2 iterations of source loop

Version on Stride: SMG2000* hypre__nx ~ 100: Rewriting the Source

```
for (loopk = 0; loopk < hypre__nz; loopk++)
    {                                                          //7
    for (loopj = 0; loopj < hypre__ny; loopj++)
            {                                                  //8
            for (loopi = 0; loopi < hypre__nx; loopi++)
                    {                                          //9
                    rp[ri] -= Ap[Ai] * xp[xi];
                    Ai += hypre__sx1; ← version on stride
                    xi += hypre__sx2;
                    ri += hypre__sx3;
                    }                                          //8
            Ai += hypre__sy1 - hypre__nx*hypre__sx1;
            xi += hypre__sy2 - hypre__nx*hypre__sx2;
            ri += hypre__sy3 - hypre__nx*hypre__sx3;
            }                                                  //7
    Ai += hypre__sz1 - hypre__ny*hypre__sy1;
    xi += hypre__sz2 - hypre__ny*hypre__sy2;
    ri += hypre__sz3 - hypre__ny*hypre__sy3;
    }                                                          //6
```

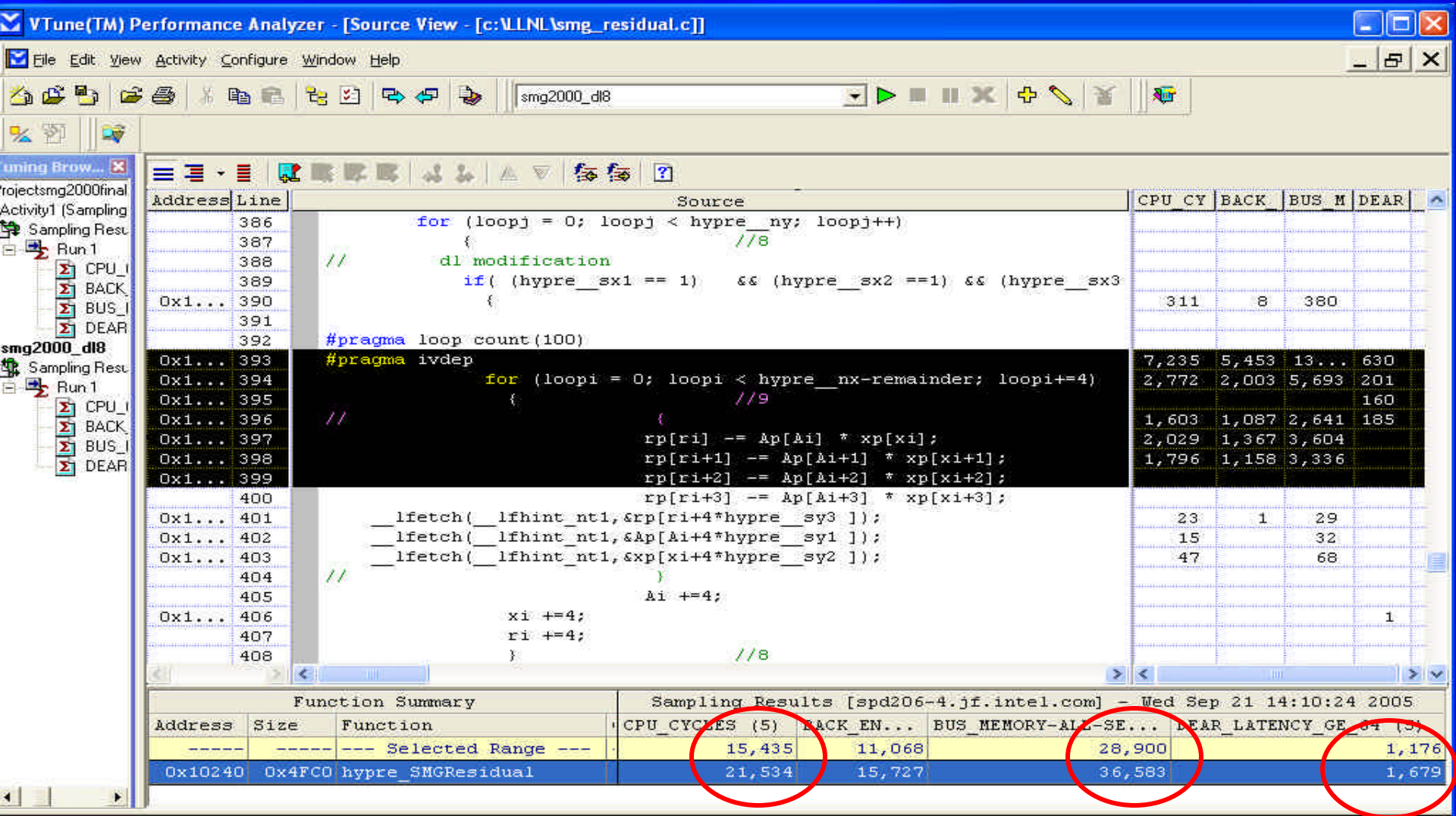
Version on Stride: SMG2000*
Version loops on the strides = 1!

```
        for (loopk = 0; loopk < hypre__nz; loopk++)
            {                                                          //7
            for (loopj = 0; loopj < hypre__ny; loopj++)
                {                                                      //8
//      version on stride
                if( (hypre__sx1 == 1)&& (hypre__sx2 ==1) && (hypre__sx3 ==1))
                 {
#pragma loop count(100)
                  for (loopi = 0; loopi < hypre__nx-remainder; loopi+=4)
                        {                                              //9
                    rp[ri] -= Ap[Ai] * xp[xi];
                    rp[ri+1] -= Ap[Ai+1] * xp[xi+1];
                    rp[ri+2] -= Ap[Ai+2] * xp[xi+2];
                    rp[ri+3] -= Ap[Ai+3] * xp[xi+3];
      __lfetch(__lfhint_nt1,&rp[ri+4*hypre__sy3 ]);
      __lfetch(__lfhint_nt1,&Ap[Ai+4*hypre__sy1 ]);
      __lfetch(__lfhint_nt1,&xp[xi+4*hypre__sy2 ]);
                        Ai +=4;      <--unroll by 4 (instead of 8) because I'm lazy
                        xi +=4;
                        ri +=4;
                        }                                              //8
//    remainder loop
```

Prefetching with Short tripcount: SMG2000*
Outer loop Driven Prefetch

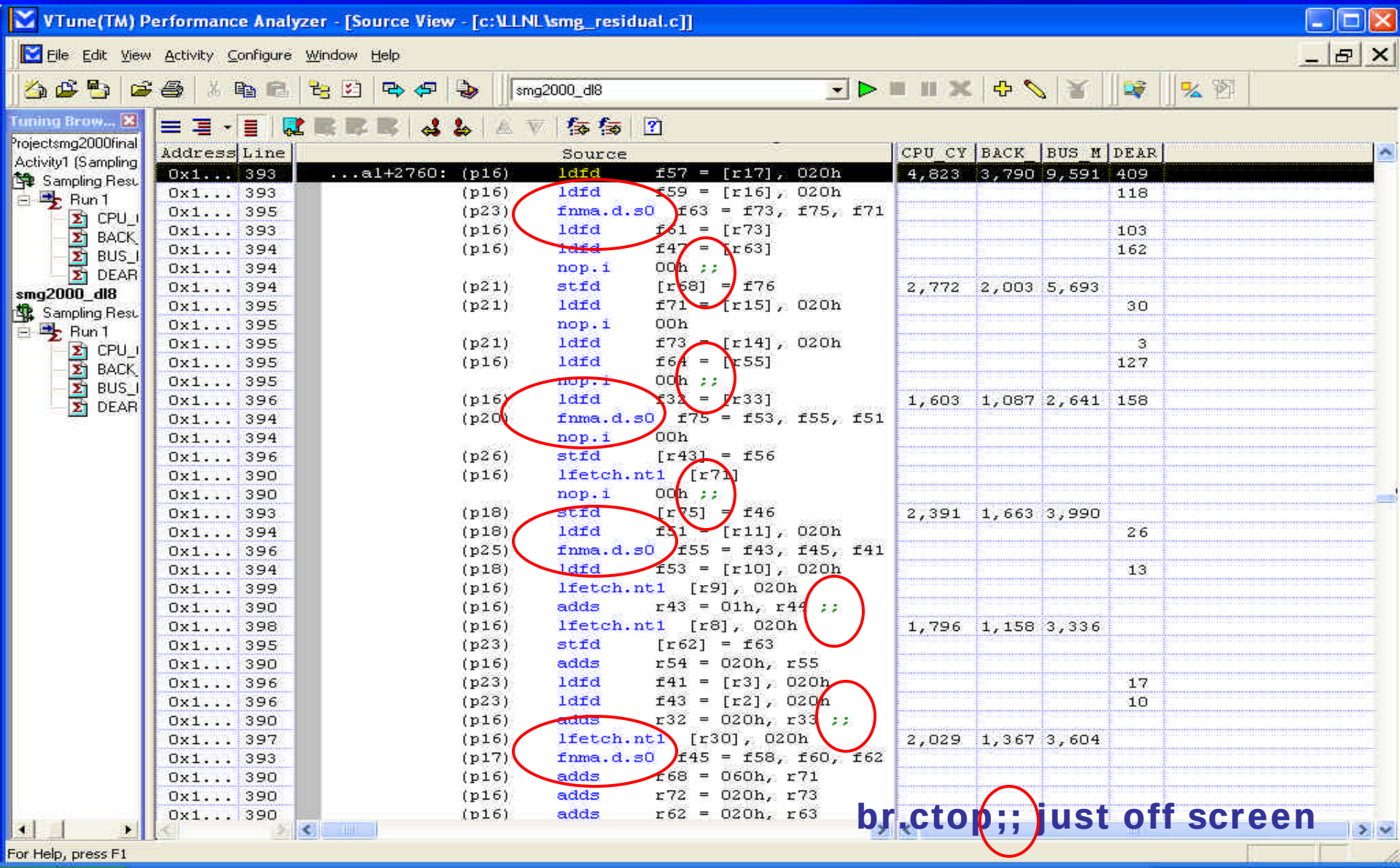
```
        for (loopk = 0; loopk < hypre__nz; loopk++)
            {                                                          //7
            for (loopj = 0; loopj < hypre__ny; loopj++)
                {                                                      //8
//      version on stride
                if( (hypre__sx1 == 1)&& (hypre__sx2 ==1) && (hypre__sx3 ==1))
                 {
#pragma loop count(100)     ←force compiler generated prefetch to be "close"
                  for (loopi = 0; loopi < hypre__nx-remainder; loopi+=4)
                      {                                                //9
                rp[ri] -= Ap[Ai] * xp[xi];
                rp[ri+1] -= Ap[Ai+1] * xp[xi+1];
                rp[ri+2] -= Ap[Ai+2] * xp[xi+2];
                rp[ri+3] -= Ap[Ai+3] * xp[xi+3];
    __lfetch(__lfhint_nt1,&rp[ri+4*hypre__sy3 ]);      ←---prefetch for loopj + 4
    __lfetch(__lfhint_nt1,&Ap[Ai+4*hypre__sy1 ]);
    __lfetch(__lfhint_nt1,&xp[xi+4*hypre__sy2 ]);
                      Ai +=4;      ←--unroll by 4 (instead of 8) because I'm lazy
                      xi +=4;
                      ri +=4;
                      }                                                //8
//    remainder loop
```

SMG2000* rewritten



Loop is Now Bandwidth Limited

SMG2000* Final Asm



4 iterations in 6 cycles

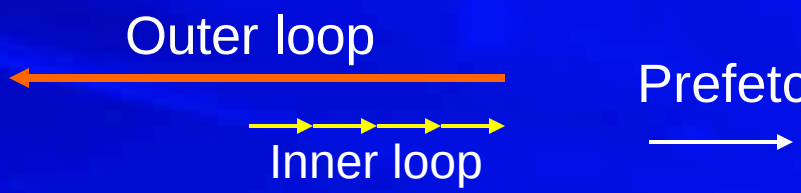
Summary

- **BE EXPLICIT: Don't do Anything Tricky**
  - **At Best You'll ONLY Confuse the Compiler**
  - **More Likely You'll Confuse Yourself AND the Compiler**
- **General solutions might be "nice"…**
  - **But they are likely to be slow**
- **Optimized Solutions take advantage of unique aspects of your problem**
- **Use all the tools in a Coordinated Manner**

Backup

Sawtooth Access Patterns are Tough on HW/SW prefetching

- In solvers, incrementing followed by decrementing sums are common.
- This results in loops going forward followed by loops going backwards.
- Backwards loops frequently have a sawtooth access pattern
  - Short inner loop going forward in address
  - Outer loop making large jumps in negative direction in address



- HW and SW prefetch tend to see inner sequence only and thus “prefetches” data that has already been used

Identifying and Resolving Sawtooth Access

- Identification
  - Usual stuff: use dear_latency_gt_64 and opcode tagged L3_read.data_read.miss
  - Inspection of SOURCE for negative loop stride
    - May require printf

Resolution:

- Change direction of inner loop
  - These loops tend to be “sum-=“ so order is irrelevant
- Lfetch intrinsics and do it by hand

Disambiguation

- C and C++ standards state that the compiler MUST assume that ambiguous pointers overlap
- Fortran standard states that arrays with different names can be assumed by the compiler to NOT overlap
  - Unless an equivalence statement is used
- For(i=0; i<len; i++)a[i] += x*b[i];
- The compiler must assume that &a[i] = & b[i+1]
  - EITHER: Preventing any loop unrolling or any parallel instruction execution
  - OR: requireing speculative execution of multiple iterations
    - Through an O-O-O engine like x86
    - Speculative instruction generation by the compiler

Ambiguous Pointers are ALWAYS BAD

- If the code generation for ambiguous pointers were better then that code sequence would be the compilers DEFAULT
- Always requires more compiler gymnastics
  - Speculative instructions
  - Versioning
  - Things it wouldn't do otherwise
- If code with ambiguous pointers runs faster..it is a bug..pure and simple
- Fortran can have ambiguous pointers due to lost (instrinsic) disambiguation (of fortran std) in the Intermediate language
  - Arrays in common block..etc

Identifying an Ambiguity Performance Problem

- Visual Inspection of Assembler is the ONLY reliable test
- Must use runtime performance data to identify code path that is actually executed
  - Non executed versions are irrelevant
- VTune™ Performance Analyzer asm view as spread sheet, exported to excel, is the best method
  - Sort by source line
  - Find hotspot's address by seeing what addresses have lots of events
  - Resort data by address and scroll down to hotspot
  - Maintain source listing view in VTune Analyzer for comparison
  - Make sure there are not multiple hotspots in asm view for a single block of source..(consistency of total event counts)

Identifying an Ambiguity Performance Problem

- Visual Inspection of Assembler is the ONLY reliable test
- Look for speculative instructions
  - If there weren't a load/store pointer ambiguity, there wouldn't be any speculation
  - ld.a, ld.s, chk
  - Lack of SWP (br.ctop)??
- "Greedy Scheduling bug"
  - Lots of loads and adds with no interspersed fma's
  - Followed by bundles with 2 fma's and 4 nops
  - Finally stores, adds, lfetches and br.ctop
  - Why weren't you scheduled at the resource ii
    - = number of fma's/2
  - Check the source and calculate the number of required fp ops

Identifying Disambiguation Performance Issues

```
void DAXPY(int len, double x, double * a, double *b){
    int i,j;
            for(i=0; i<len; i++)a[i] += x*b[i];
    return;}
(p16)ldfpd          f37,f34=[r36]
(p16)ldfd.a         f32=[r3],16
(p17)fma.d          f40=f8,f33,f38
(p16)lfetch.nt1     [r34] ;;
(p18)stfd           [r2]=f39,16
    nop.i 0 ;;
(p17)chk.a.nc       f33,.b1_78
    nop.m           0
    nop.i 0
(p17)stfd           [r37]=f40
(p17)ldfd           f39=[r9],16
    nop.i 0 ;;
(p16)add            r35=16,r36
(p17)fma.d          f38=f8,f39,f35
(p16)add            r32=32,r34
    nop.m           0
    nop.i 0
    br.ctop.sptk    .b1_18 ;;
```

Ld.a & chk.a means there is an ambiguity

Note: 9.0 compiler “versions” around this example

Disambiguation Resolution

- -fno-alias, -ansi_alias (Linux) –Oa(windows)
- -restrict
  - + restrict keyword in source

```
void DAXPY(int len, double x, double * restrict a, double *b)
{
    int i;
    for(i=0; i<len; i++)a[i] += x*b[i];
    return;
}
```

- Restrict is NOT defined for C++
  - What does it even mean to apply restrict to a class?

Function Calls Exporting Pointers can Cause Ambiguities

2D Arrays declared with ** notation

While(norm > tolerance){

```
MPI_Send(top_row[]….);
MPI_Send(bottom_row[]…);
MPI_Recv(bottom_of_top_neighbor[]….);
MPI_Recv(top_of_bottom_neighbor[]…);
For(i=1;i<dim1;i++)
      For(j=1;j<dim2;j++)new[i][j] = (old[i+1][j] + old[i-1][j] +
                                    old[i][j+1] + old[i][j-1] +src[i][j] )*0.25;
dsum = 0;
For(i=1;i<dim1;i++)
      For(j=1;j<dim2;j++)
                  dsum += (new[i][j] - old[i][j])* (new[i][j] - old[i][j]); ←----- ambig *
MPI_Allreduce(&dsum, &norm…..); ←------- export &dsum
For(i=1;i<dim1;i++)
      For(j=1;j<dim2;j++)old[i][j] = new[i][j];
}
```

Export of &dsum allows local variable dsum to become Ambiguous with **new and **old

Function Calls Exporting Pointers can Cause Ambiguities

2D Arrays declared with ** notation

```
dsum = 0;
For(i=1;i<dim1;i++)
    For(j=1;j<dim2;j++)
            dsum += (new[i][j] - old[i][j])* (new[i][j] - old[i][j]); ←-----
      ambig *
    tsum = dsum ←----------
MPI_Allreduce(&tsum, &norm…..); ←------- export &dsum
```

Introduce Local Copy to Distinguish Addresses