Mysteries of Performance Analysis

David F. Bacon

(joint work with Peter F. Sweeney,
K. Muthukumar, and Vivek Sarker)

The Case of the Excess Speedup

Virtual Function Resolution

- Converts C++ indirect calls to direct calls
- Speedups expected: 0 - 35%
  - depending on virtual function usage
- PROBLEM: wildly inconsistent speedups

Variation in Speedup from VFR

SPEEDUP		Static Linking	Dynamic Linking
	RS/6000 520	19%	10%
	RS/6000 590	2%	20%

- What is effect of differences:
  - in architecture?
  - in linkage?
- What architectural features are especially bad/good for virtual function calls?

Theories

- Branch prediction/prefetch
- PTRGLUE or virtual function tables were on conflicting cache or TLB lines
- Shrinking of code eliminated cache or TLB line conflicts
- Unknown weirdness with dynamic linking

Theory 1: Branch Prediction

- Nutshell: resolved branches improve prefetch
- Con:
  - should perform the same (relatively) regardless of linkage
  - VFR removes 7 loads, 1 store, and 11 instructions; prefetch benefits should be dwarfed by this

Theory 2: ptrglue/vtable conflicts

- Nutshell: ptrglue code or a vtable is conflicting with another TLB or cache line
- Con:
  - Many different vtables; probability lower
  - Only some virtual calls resolved
    - » ptrglue page should remain hot

Theory 3: Code shift conflicts

- Nutshell: VFR reduces size of function bodies; shifts code to remove cache/TLB conflict
- Con:
  - should be more random across programs
  - code actually 750 bytes larger after VFR
  - probability of occuring is extremely low

TLB Conflict Probabilities

- code is 800K statically linked = 200 pages
- I-cache /I-TLB 2-way set-associative
- 64 TLB sets; max 3 conflicting pages
- Assume 90/10 rule: 20 “hot” pages
- Probability of 3 hot pages mapping to one set is extremely low

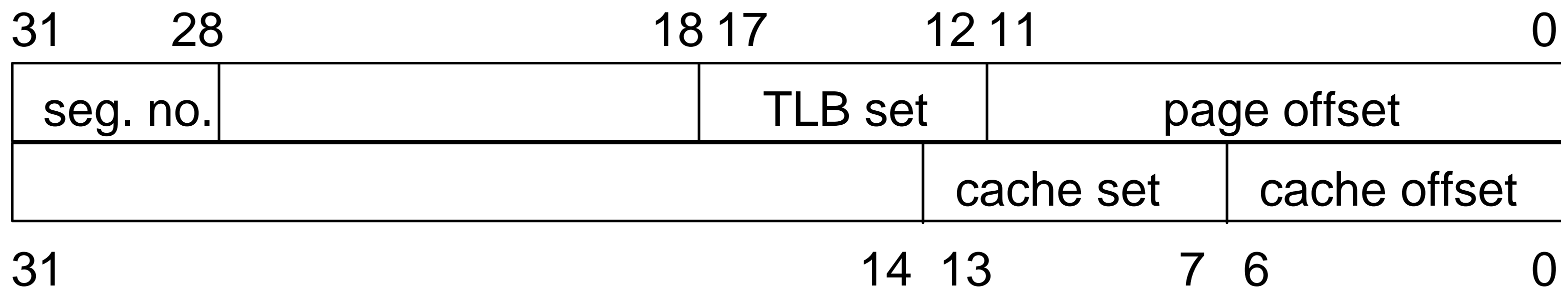
Performance Monitor Measures

	Unoptimized	Optimized (VFR)
I-Cache Misses	0.806	0.604
D-Cache Misses	0.149	0.122
I-TLB Misses	2.120	0.017
D-TLB Misses	2.184	0.082
Cycles No Disp.	167.970	125.400

PV Measurements

- 3 "hot" pages conflict in unoptimized prog:
  - 4th page in kernel, user, and library segments
- library page is libc.a:malloc()
- Code reordered due to VFR

Diagnosis: Randomness Reduced



- Segment layout always starts at page 0
- As TLB's get larger and segments increase, underpopulated segments more likely
- Result: TLB randomness principle destroyed

Solutions

- Increase hardware associativity of TLB
- Re-balance page offsets in software
  - profile hot pages in kernel and libc
  - have loader fill those pages last
- HeatShrink must not start layout at page 0

The Case of the Mysterious Padding

Cache/TLB Padding Selection

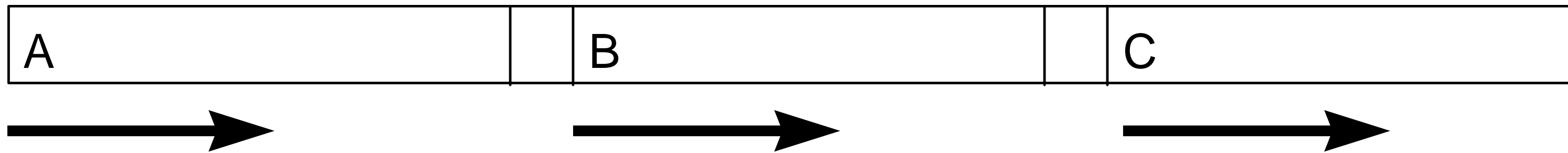
- Problem: Gemstone compiler's array padding algorithm
- SPECfp swm256 5% slower than w/VAST
  - only difference is inter-array padding
  - difference narrowed down to one inner loop

A Closer Look at the Loop

```
REAL*4 A(257,257), B(257,257), C(257,257),...

do j
    do i
        ...A[i,j]...
        ...A[i+1,j]...
        ...B[i,j+1]...
        ...C[i+1,j+1]...
        ...
    end do
end do
```

Padding



- 7 arrays; 14 references
- Sequential, synchronized access of arrays
- Exceeds D-cache size
  - main goal of padding: minimize conflicts

The Actual Loop

```
REAL*4 U(257,257),V(257,257),P(257,257),CU(257,257),CV(257,257),
       Z(257,257),H(257,257)

DO 100 J=1,N
   DO 100 I=1,M
     CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
     CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
     Z(I+1,J+1)= (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*
                  (U(I+1,J+1)-U(I+1,J)))/(P(I,J)+P(I+1,J)+
                   P(I+1,J+1)+P(I,J+1))
     H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*
              U(I,J)+V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
100 CONTINUE
```

Padding Selection

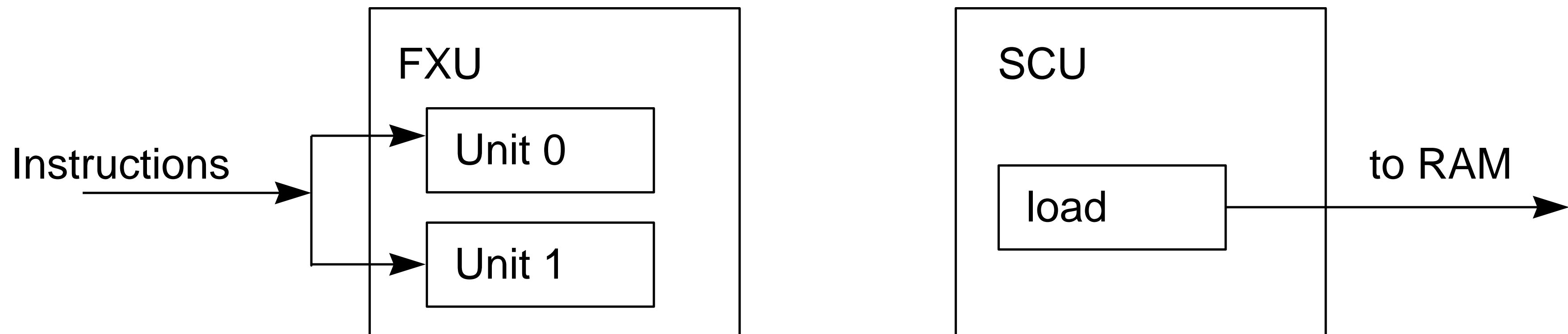
- Inter-array padding:
  - VAST selects 2068
  - Gemstone selects 2044
- Analytically: no conflicts

Performance Monitor Measures

	Pad=2044	Pad=2068
Cycles FXU unit 0 held b.c. D-cache busy	9.8%	3.6%
Cycles FXU unit 1 held b.c. D-cache busy	16.6%	8.6%
Cycles FPU unit 0 held by load/store instructions	10.4%	3.7%
Cycles FPU unit 1 held by load/store instructions	11.4%	5.5%

- D-cache and D-TLB misses almost identical
- What causes extra hold cycles?

Diagnosis: Synchronized Misses



- Power2 can execute during a cache miss
- But 2 misses freeze the FXU
- Maximum performance when misses are staggered, one per iteration

Cure

- Software: use padding to minimize synchronized misses
  - minimize TLB misses
  - minimize cache misses
  - minimize synchronous cache misses
- Hardware
  - bigger load queue
  - decouple FXU from load/store operations