Combining Static and Dynamic Approaches to Model Loop Performance in HPC

Vincent Palomares

Ph.D. Advisor: Prof. William Jalby

UVSQ

September 21st 2015

Supercomputers

Typical Uses

- Aerodynamics
- Engine design
- Weather forecasting

Characteristics

- Great for parallel workloads
- Expensive to build
- Expensive to upkeep
- Performance matters!
  => Performance analysis and optimization

Loop Analysis

Pareto Principle for Optimization / Modeling (HPC)

- 90% time in 10% of code
- => Focus on loops

Binary Loops

- Fine-grain approach
- What is really executed
- Can be extracted
- (Individual error $< X\%$) $=>$ (cumulative error $< X\%$)

Analysis Approaches

Dynamic: Sampling

- Loop identification
- Top-down approach, stalls counters, ...
- Qualitative metrics
- Low cost / overhead ($\sim$ normal runtime)

Dynamic: Differential Analysis

- E.g. DECAN
- Patch loops, run again, compare
- Quantitative metrics
- More expensive (few times)

Analysis Approaches

Static Analysis

- E.g. Code Quality Analyzer (CQA)
  Intel Architecture Code Analyzer (IACA)
- Qualitative and quantitative metrics (L1)
- Cost-efficient

Full Cycle-Accurate Simulation

- Perfect knowledge and predictive ability
- Extremely expensive (millions of times)

Presentation of Code Quality Analyzer (CQA)

```
Unroll factor: 1 or NA

***************************************
              Back-end
***************************************
       P0      P1     P2   P3   P4   P5
FU     FP x/÷  FP +   LD1  LD2  ST   OTH.
Uops   18.00   17.00 9.50 9.50 3.00 6.00
Cycles 43.00   17.00 9.50 9.50 3.00 6.00

Cycles executing div or sqrt
instructions: 20-43 (second value used
for L1 performances)
Longest recurrence chain latency
(RecMII): 3.00

*************************************
        Vectorization ratios
*************************************
All      : 0%
Load     : 0%
Store    : 0%
```

```
Mul      : 0%
add_sub  : 0%
Other    : 0%

*************************************
       Vector efficiency ratios
*************************************
All      : 25%
Load     : 25%
Store    : 25%
Mul      : 25%
add_sub  : 25%
Other    : 25%

*************************************
         If all data in L1
*************************************
cycles: 43.00
FP operations per cycle: 0.81 (GFLOPS
at 1 GHz)
(…)
Cycles if fully vectorized: 21.50
```

Overview

- Fast out-of-context static analysis
- Upper bound on performance
- Distribution of the workload

DECAN Loop Transformation

Original code (REF)

```
MOVAPS   (%RAX,%RCX,8),%XMM2
MOVAPS   0x10(%RAX,%RCX,8),%XMM3
MULPD    %XMM1,%XMM2
MULPD    %XMM1,%XMM3
MOVAPS   %XMM2,(%RAX,%RCX,8)
MOVAPS   %XMM3,0x10(%RAX,%RCX,8)
ADD      $0x4,%RCX
CMP      %R11,%RCX
JB       .LOOP
```

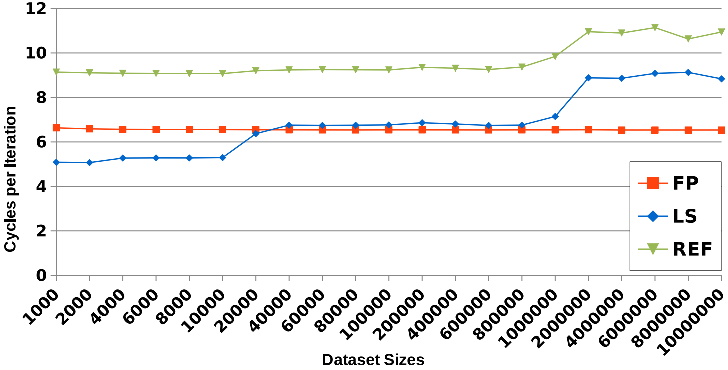
LS

```
MOVAPS   (%RAX,%RCX,8),%XMM2
MOVAPS   0x10(%RAX,%RCX,8),%XMM3


MOVAPS   %XMM2,(%RAX,%RCX,8)
MOVAPS   %XMM3,0x10(%RAX,%RCX,8)
ADD      $0x4,%RCX
CMP      %R11,%RCX
JB       .LOOP
```

FP

```
XORPS    %XMM2,%XMM2
XORPS    %XMM3,%XMM3
MULPD    %XMM1,%XMM2
MULPD    %XMM1,%XMM3


ADD      $0x4,%RCX
CMP      %R11,%RCX
JB       .LOOP
```

Example Results



Comments

- Codelet: toeplz_4_de (Numerical Recipes)

Presentation of Cape

Purpose

- Hardware / software codesign
- Performance analysis
- Extremely fast

Overview

- Loop-centric model
- Models potential bottlenecks individually (*nodes*)
- Combines small models' results
- Dozens of loops simultaneously
- Uses DECAN, CQA

Cape Nodes

Node

- Small linear model
- Can target hardware (HW) or software (SW)
- E.g. Floating point (FP) multiply, divisions, memory performance...

Workload

- Workload for a node
- E.g. 10 multiplications per loop iteration

Bandwidth

- Work processable per cycle
- E.g. 2 multiplications per cycle

Cape Modeling

Node Time

- $Time = Workload \ / \ Bandwidth$
- E.g. 5 cycles = 10 multiplications / 2 multiplications per cycle

Cape Time

- Perfect parallelism assumption
- $Time = Max_{all\ nodes}\,(node\ time)$

Accuracy

- $System\ Saturation = Cape\ time\ /\ Measured\ time$
- Ideally, $System\ Saturation = 1$

Contributions

Cape Extension

- More nodes
- Finer node modeling

VP3

- Leverages Cape
- Predicts impact of vectorization
- Helps SW optimization

Uop Flow Simulation

- Tackles lack of saturation
- Blends static analysis and simulation
- Accounts for impact of out-of-order (OoO) resources

Front-End Modeling

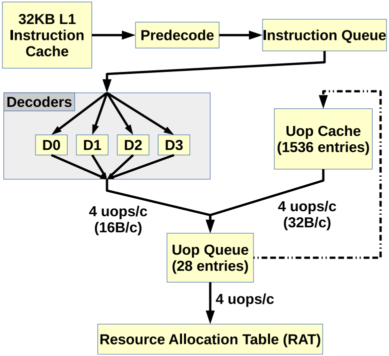
Big Core Background Information

- Can issue 4 uops per cycle
- Number of uops per instruction is known (Agner Fog)

Naive Node Implementation

- *Workload = Front-End (FE) uops*
- *Bandwidth = 4*

Front-End



Features

- Macrofusion
- Microfusion
- Unlamination
- Loop Stream Detector (LSD) limitation

Finding Unlamination Rules

Codelet	Measured		Unlamination Criteria			
	SNB2	*HSW*	**>= 3 regs.**	**>= 3 regs., except stores**	**>= 4 regs.**	**Never**
elmhes_10_de	26	18	**26**	22	*18*	*18*
elmhes_10_dx	34	30	**34**	*30*	*30*	22
svdcmp_6_dx	49	49	***49***	***49***	***49***	41
svdcmp_11_dx	8	8	***8***	***8***	***8***	***8***

Comments

- Used *UOPS_ISSUED.ANY* HWC for measurements
- Testing different criteria:
  - Blue: matches criterion on Sandy Bridge (SNB)
  - Red: matches on Haswell (HSW)
  - Purple: matches on both
- => Bigger RAT uops in HSW

FE Node Implementation

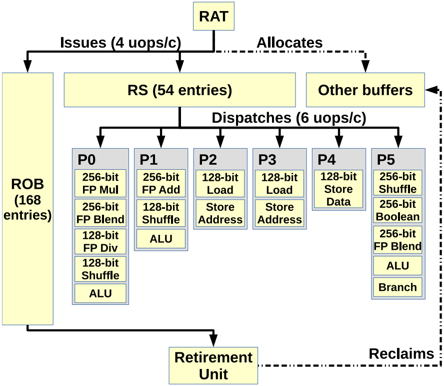
Naive

- *Workload = FE uops*
- *Bandwidth = 4*

Improved

- Derived from previous observations
- *Workload = ceiling (post-unlamination uops / 4) * 4*
- *Bandwidth = 4*

Back-End



Features

- In-order issue
- Out-of-order dispatch (OoO)
- In-order retire

FP Adder Modeling

Operation	Assigned Workload							
	Add node 1	Add node 2	Add node 3	Add node 4	Add node 5	Add node 6	Add node 7	Add node 8
256-bit vector add	1	1	1	1	1	1	1	1
128-bit vector add	1	1	1	1				
DP scalar add	1	1						
SP scalar add	1							
	Physical adder length (256-bit)							

Comments

- Several nodes to model FP FUs
- Workload as described above
- Each node's BW = 1
- Finer grain view and control

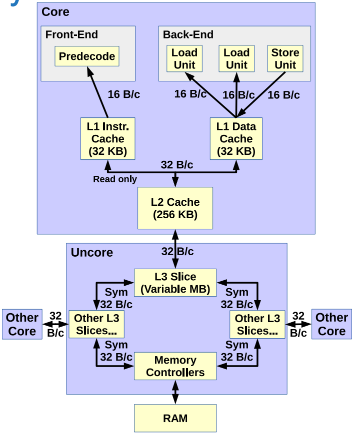
Store Modeling

Operation	Assigned Workload			
	Store node 1	Store node 2	Store node 3	Store node 4
256-bit vector store	2	2	2	2
128-bit vector store	1	1	1	1
DP scalar store	1	1		
SP scalar store	1			
	Store data path length (128-bit)			

Comments

- Same as for FP adder modeling
- Functional unit (FU)'s width $<$ vector size on SNB
  $- >$ Fixed in HSW

Memory Hierarchy



Features

- Dedicated L1 and L2
- Distributed L3

Post-L1 Cache Modeling

Challenge

- Memory performance is fickle
- Stride, number of streams, instruction types

Approach

- Nodes for L2, L3 and RAM traffic
- Workload = transferred cache lines using hardware counters (HWC)
  E.g. on SNB:
  L2 node workload = L1D.REPLACEMENT + L1D_WB_RQST.ALL
- Bandwidth obtained with DECAN LS variant:
  $$BW = max_{across\ runs}\left(\frac{workload}{cycles}\right)$$

Conclusion

Improvements

- Handles more components
- Better system saturation
- Finer-grain view and control

Context

Vectorization

- Hardware/software optimization
- Single Instruction, Multiple Data (SIMD)
- Up to 8x speedup (soon 16x)
- Can be hard to use

VP3

- Leverages Cape
- Projects vectorization potential
- Disregards legality

Usecase (POLARIS Loop)

```
do j = ni+nvalue1,nato
    nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
    u1 = x11 - x(nj1) ; u2 = x12 - x(nj2) ; u3 = x13 - x(nj3)
    rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
    rij = demi*(rvwi+rvwalc1(j))
    drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
    Eq = qq1*qq(j)*drtest
    ntj = nti + ntype(j)
    Ed = ceps(ntj)*drtest2*drtest2*drtest2
    Eqc = Eqc + Eq ; Ephob = Ephob + Ed
    gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
    u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
    g1c = g1c - u1g ; g2c = g2c - u2g ; g3c = g3c - u3g
    gr(nj1,thread_num) = gr(nj1,thread_num) + u1g
    gr(nj2,thread_num) = gr(nj2,thread_num) + u2g
    gr(nj3,thread_num) = gr(nj3,thread_num) + u3g
end do
```

Variable number of iterations

Non-unit stride accesses

DIV/SQRT

Indirect accesses

Reductions

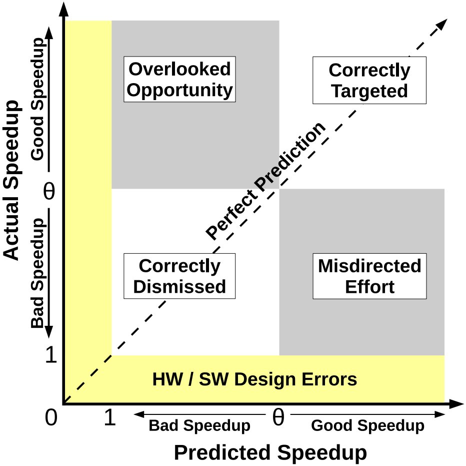
Non-unit stride accesses

High number of statements vector versus calar

Comments

- Many possible problems
- Is vectorizing worth the trouble?

Operating Space of Performance Prediction



Comments

- θ: minimum speedup for user consideration

Projection Steps

Steps

1. Get regular Cape input:
    - LS measurements
    - FP CQA analysis
2. CQA vectorization mockup (FP)
3. Scale memory node bandwidths
4. Adjust Cape saturation (LS/FP balance)

CQA Vectorization Mockup

Mockup Generation

- Unroll
- Group scalar instructions into packs
- Replace scalar packs with vector instructions

CQA Analysis

- Normal analysis on mockup loop
- Use new result values as Cape input

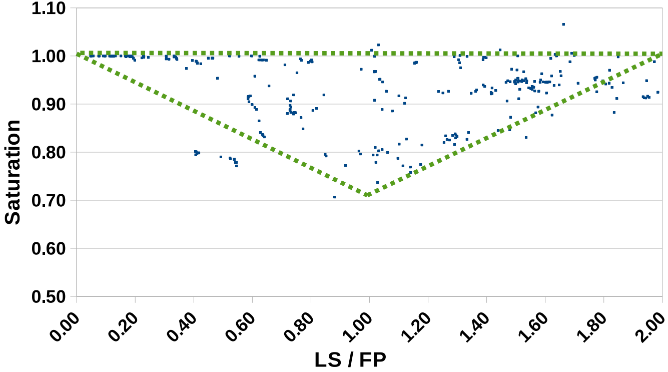
Bandwidth Improvements due to Vectorization

Type of Scalar	L2 Speedup	L3 Speedup	RAM Speedup
Single Precision	2.04	1.63	1.25
Double Precision	1.79	1.40	1.10

Comments

- Obtained with microbenchmarks
- Machine-dependent
- Only for full vectorization

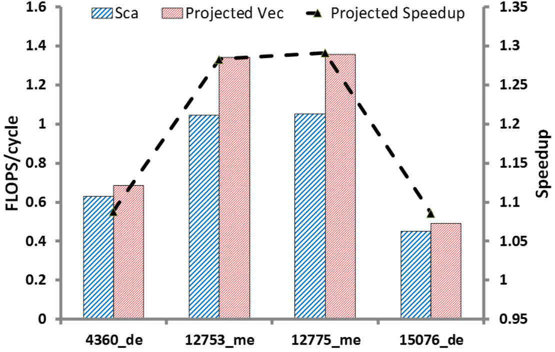
Handling Imperfect Saturation



DECAN-level Saturation

- $\simeq$ 550 data points (NRs)
- Correlation between saturation and $LS\ /\ FP$
- Can be used to adjust Cape projections

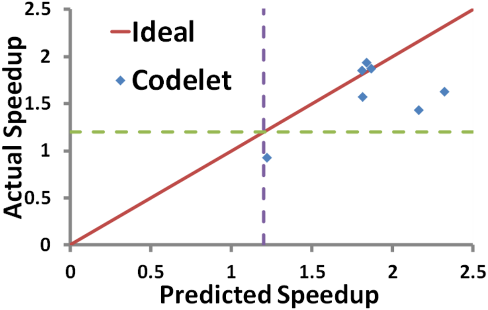
Vectorization Potential for YALES2



Comments

- Turbulent reactive flow (Computational Fluid Dynamics)
- Used by Areva, Safran
- Low prospects for vectorization

Vectorization Potential for POLARIS(MD)



Comments

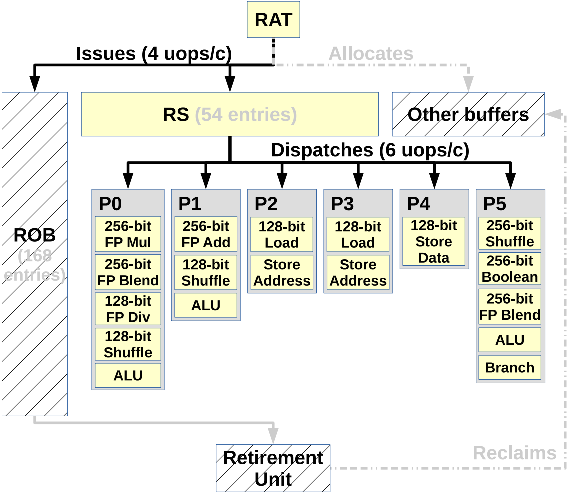
- Molecular Dynamics (by CEA DSV)
- Projecting from scalar code
- Comparing to real vector code
- 6 good results, 1 bad (too optimistic)

Conclusion

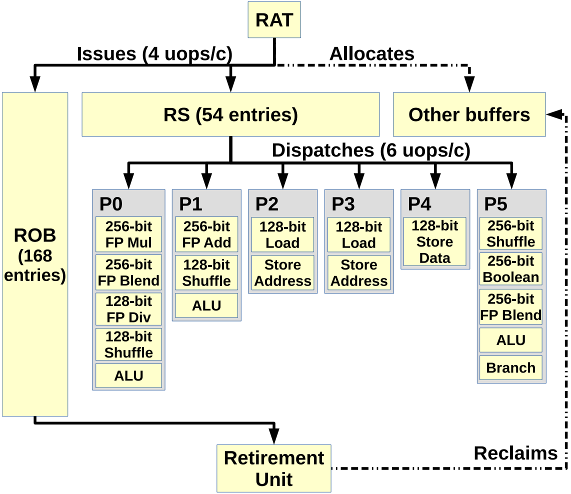
VP3

- Novel approach
- Quality projections
- Helps optimization process
- Showcases Cape model

CQA View of the Control Unit (SNB)



What if Everything were Modeled?



Accuracy Metrics

Formulas

- $Error = abs\left(\frac{measured\ time\ -\ predicted\ time}{measured\ time}\right)$
- $Fidelity = 1 - error$

Motivating Example: Realft2_4_de

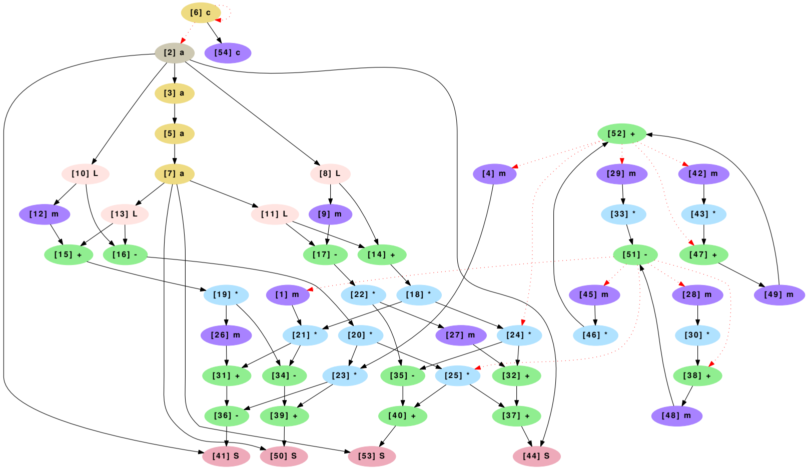
Presentation

- Codelet from the Numerical Recipes
- Part of reverse Fourier transform
- Many FP operations
- Many dependencies

Source Code

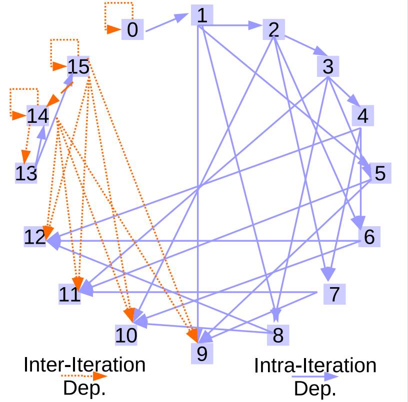
```
do i=3, ishft (n, -2) + 1
    i1 = i + i - 1
    i2 = 1 + i1
    i3 = np3 - i2
    i4 = 1 + i3
    h1r = c1 * (dat (i1) + dat (i3))
    h1i = c1 * (dat (i2) - dat (i4))
    h2r = -c2 * (dat (i2) + dat (i4))
    h2i = c2 * (dat (i1) - dat (i3))
    dat (i1) = h1r + wr * h2r - wi * h2i
    dat (i2) = h1i + wr * h2i + wi * h2r
    dat (i3) = h1r - wr * h2r + wi * h2i
    dat (i4) = -h1i + wr * h2i + wi * h2r
    wtemp = wr
    wr = wtemp * wpr - wi * wpi + wr
    wi = wi * wpr + wtemp * wpi + wi
end do
```

Assembly DDG



Source Code and DDG

```
0   do i=3, ishft (n, -2) + 1
1       i1 = i + i - 1
2       i2 = 1 + i1
3       i3 = np3 - i2
4       i4 = 1 + i3
5       h1r = c1 * (dat (i1) + dat (i3))
6       h1i = c1 * (dat (i2) - dat (i4))
7       h2r = -c2 * (dat (i2) + dat (i4))
8       h2i = c2 * (dat (i1) - dat (i3))
9       dat (i1) = h1r + wr * h2r - wi * h2i
10      dat (i2) = h1i + wr * h2i + wi * h2r
11      dat (i3) = h1r - wr * h2r + wi * h2i
12      dat (i4) = -h1i + wr * h2i + wi * h2r
13      wtemp = wr
14      wr = wtemp * wpr - wi * wpi + wr
15      wi = wi * wpr + wtemp * wpi + wi
16  end do
```



Measurements & Results for REF

Metric	Cycles per Iteration	Fidelity
Measured	23.36	N/A
CQA	16.00	68.49%
UFS (normal buffers)	23.01	98.50%
UFS (large buffers)	19.03	N/A

Observations

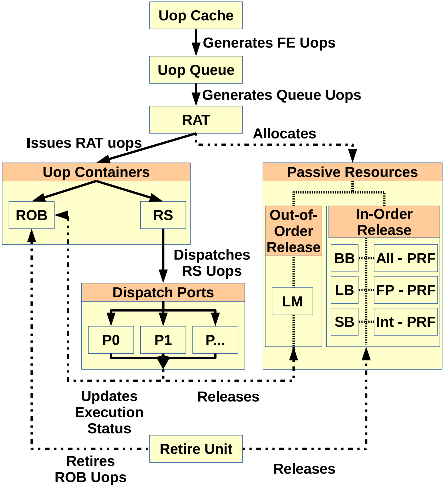
- Accurate UFS result
- Can precisely quantify RS's size impact
- UFS (large buffers) $\neq$ CQA
  Time lost due to dispatch order

UFS Model Overview

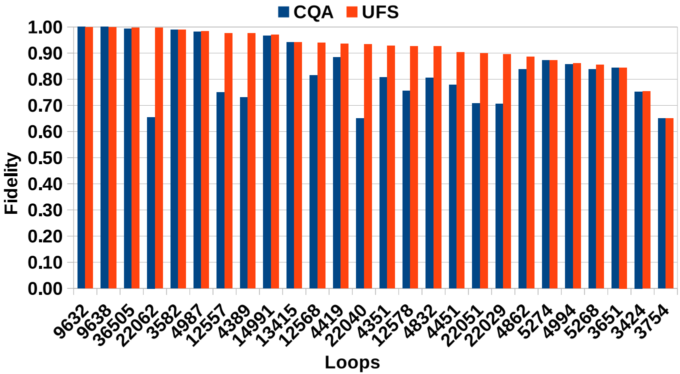
General Principles

- Ignore semantics (assume L1)
- Out-of-context analysis
- Decompose instructions into uops
- Cycle-accurate simulation of core pipeline

UFS Model Overview



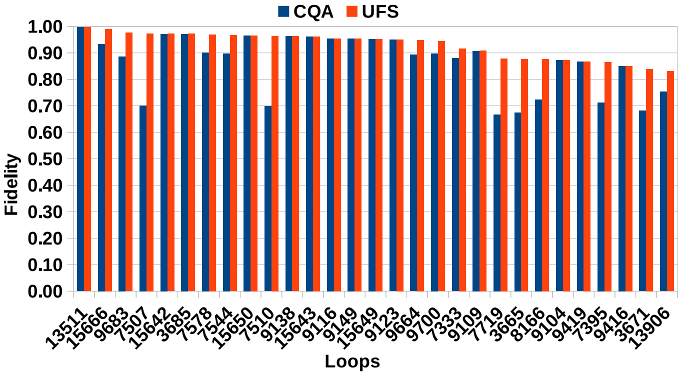
Precision Gains (YALES2: 3D Cylinder [Areva, Safran])



Notes

- Numerical simulator (CFD, turbulent flows)
- DL1 DECAN variant
- Important gains (ROB)

Precision Gains (AVBP [Alstom, Safran])



Notes

- Numerical simulator (CFD, reactive unsteady flows)
- DL1 DECAN variant
- Important gains (ROB, RS)

Speed (1000 iterations)

YALES2 (3D Cylinder)

- 110 assembly statements / loop body
- $\sim$ 8 loops / second
- 281k cycles per second

AVBP

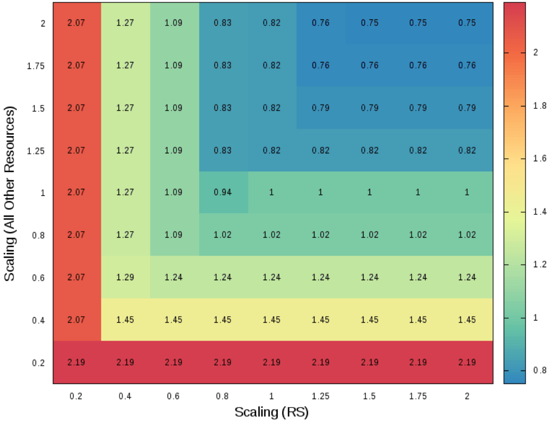
- 200 assembly statements / loop body
- $\sim$ 3.5 loops / second (5 without outlier)
- 300k cycles per second

Speed (1000 iterations)

UFS vs. CQA

- $\sim$ 5 times slower for YALES2 hotspots
- $\sim$ 9 times slower for AVBP hotspots
- $\sim$ 15 times slower for simple loops
- $\sim$ 10 times slower overall

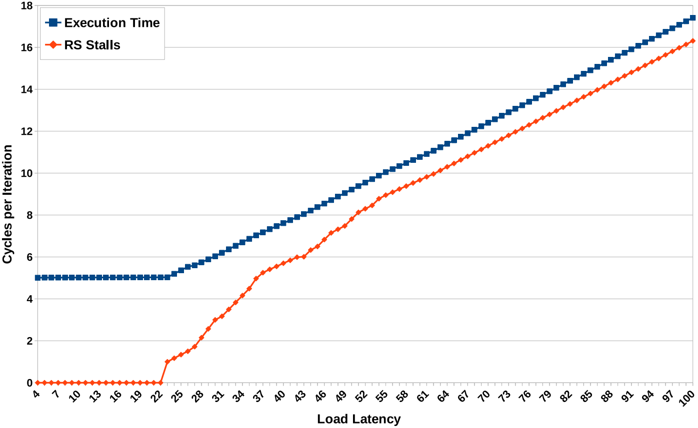
Sensitivity Analysis (OoO Resource Sizes)



Comments

- Loop 4389, YALES2
- Showing time scaling (lower is better)

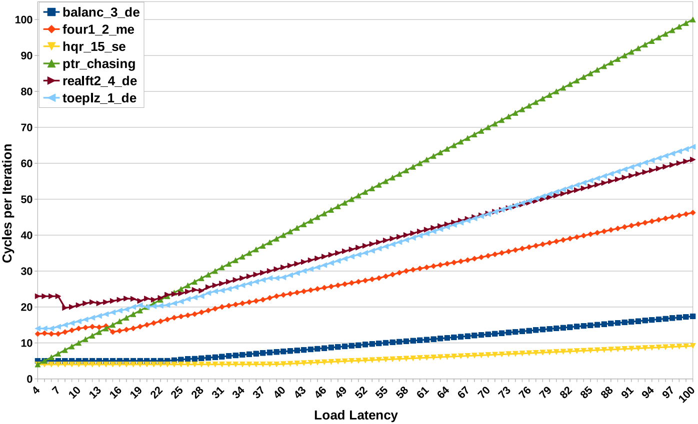
Sensitivity Analysis (Load Latency)



Comments

- Balanc_3_de (Numerical Recipes)

Sensitivity Analysis (Load Latency)



Comments

- Numerical Recipe codelets (except pointer chasing)
- Different slopes

Conclusion

Blending Approaches

- Static analysis / simulation
- Up to 35 % point gains (L1)
- Fast (10 loops / second)

Quick Insights

- Impact of OoO resources
- Impact of dispatch algorithm
- Impact of latency

Contributions

Cape

- Combines sampling, static analysis and differential analysis
- Better precision
- Finer-grain model / control

VP3

- Good quality projections
- Helps optimization process
- Cape validation

Contributions

UFS

- Tackles problems encountered with Cape and CQA
- Combines static analysis and cycle-accurate simulation
- Out-of-order engine modeling

Future Work

Cape

- More nodes
- Model latency

VP3

- AVX-512
- More validation scenarios for Cape

UFS

- More HW details (e.g. impact of stores on dispatch)
- More uarchs (Broadwell, Skylake, Silvermont...)
- Couple with dyn. information (misses, etc.)
- Integration with Cape, MAQAO

Acknowledgments

Tools and Data

- In vivo measurements (E. Oseret and M. Tribalat)
- CQA (E. Oseret)
- DECAN (Z. Bendifallah and M. Tribalat)
- Cape Tool (D. Kuck and D. Wong)

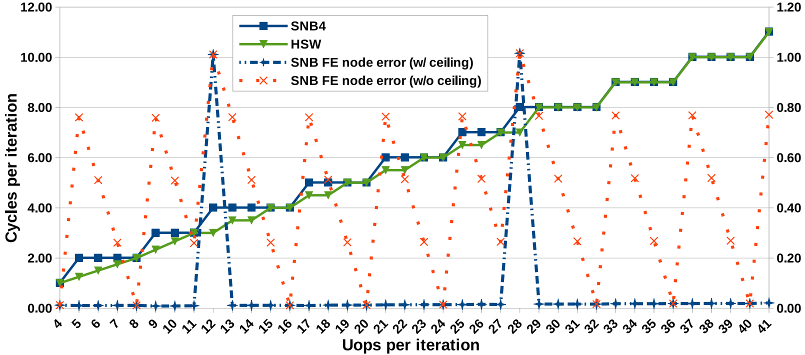
Applications

- AVBP (G. Staffelbach)
- POLARIS(MD) (M. Masella)
- RTM (H. Calandra and A. Farjallah)
- YALES2 (G. Lartigue and V. Moureau)

Thanks for your attention!

Extra Slides

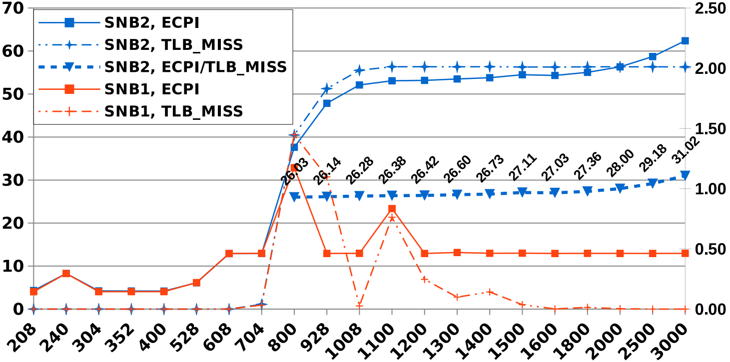
Exposing the FE Ceiling Effect



Comments

- Uops from different iterations cannot be issued together
- Fixed in HSW
- Spikes due to failed macrofusion

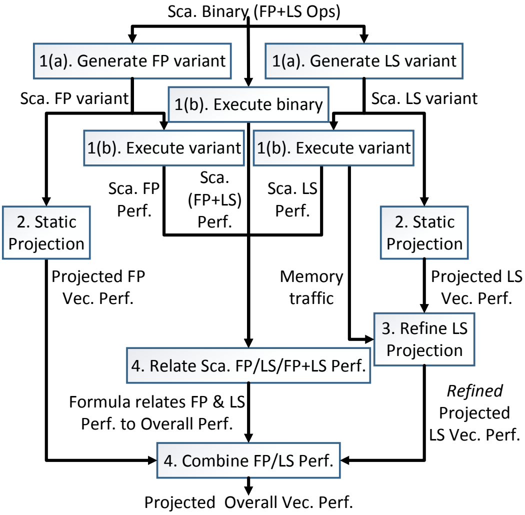
Translation Lookaside Buffer (TLB) Misses



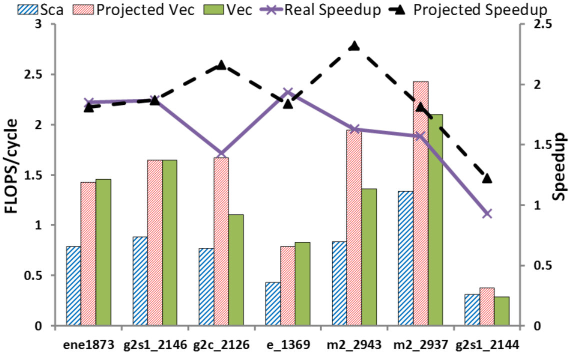
Comments

- Using HWC to count TLB misses (and get workload)
- SNB1 uses Transparent Huge Pages
- $BW = 1/26$ (for SNB2)

Projection Steps



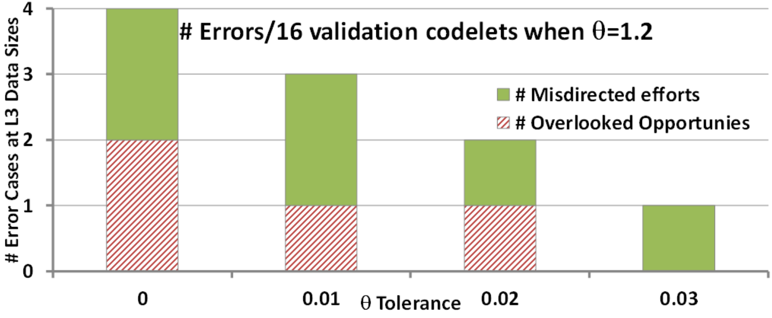
Validation Results for POLARIS



Comments

- Molecular Dynamics
- Developed by CEA DSV
- Measured both scalar and vector codes
- Projecting from scalar code

Validation Results for NRs



Comments

- Good results overall
- 3 borderline cases
- 1 bad case

SNB Measurements (L1)

Metric	REF	LS	FP	FES
Measured Duration	23.36	5.21	20.21	16.26
CQA Duration Projection	16.00	5.00	16.00	14.5
Error	31.51%	4.03%	20.83%	10.82%
Measured Resource Stalls	7.85 [RS]	0.01	7.31 [RS]	0.01

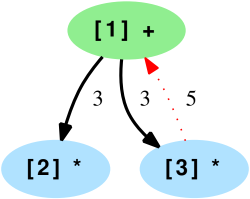
DECAN Variants

- REF: original code; LS: only loads and stores
- FP: only arithmetic instructions; FES: instructions converted to NOPs

Observations

- Important CQA gap
- $REF > max(LS, FP, FES)$ (for time and stalls)
- $Measurement - stalls < peak\ theoretical\ perf.$ (for REF and FP)

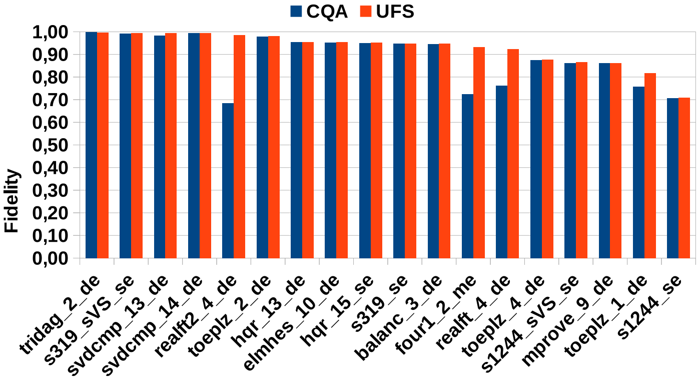
Dispatch Conflict Example



Notes

- Inter-iteration dependency $=>$ at least 8 cycles per iteration
- Pseudo-FIFO picks uop [2], but suboptimal
- Delays the inter-iteration dependency resolution by 1 cycle
- Picking uop [3] would be better, but more complex

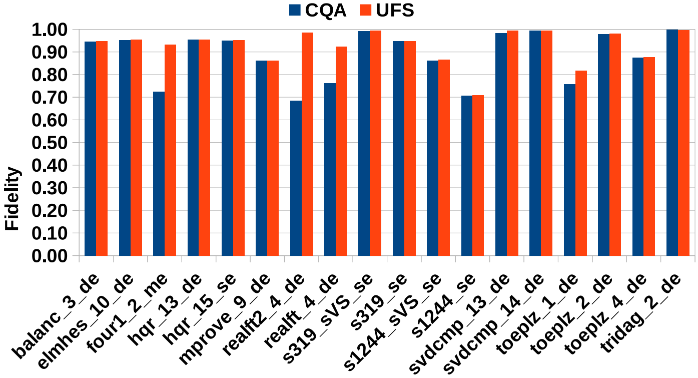
Precision Gains (NRs, REF, L1)



Notes

- Important but localized gains (RS)
- Some codelets still not understood

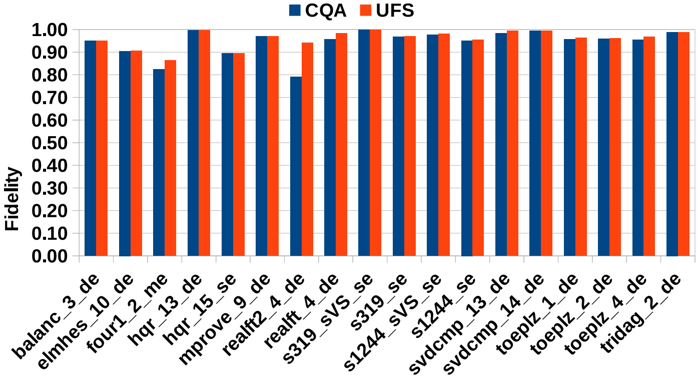
Precision Gains (NRs, REF, L1)



Notes

- Alphabetical order
- Important but localized gains (RS)
- Some codelets still not understood

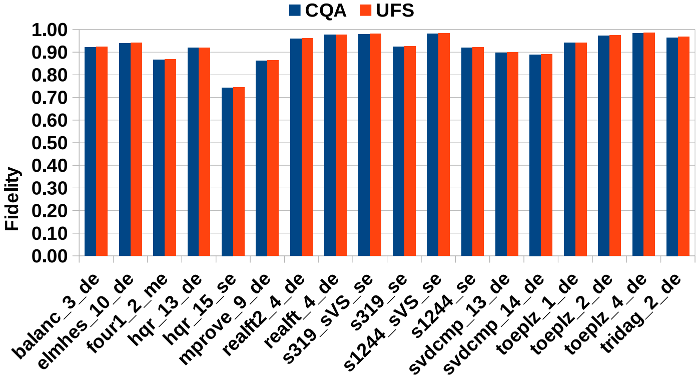
Precision Gains (NRs, FP)



Notes

- Alphabetical order
- Little gain

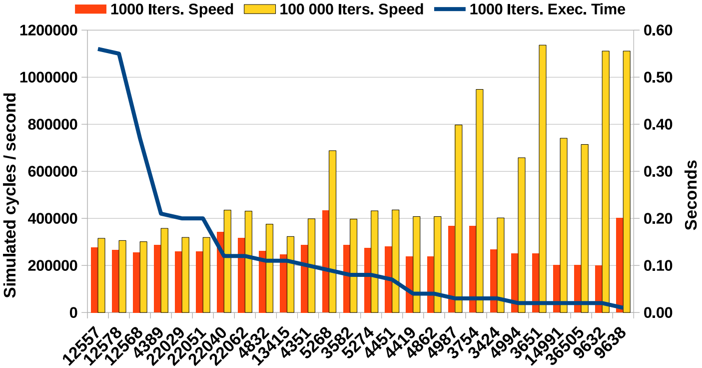
Precision Gains (NRs, LS, L1)



Notes

- Alphabetical order
- No gain

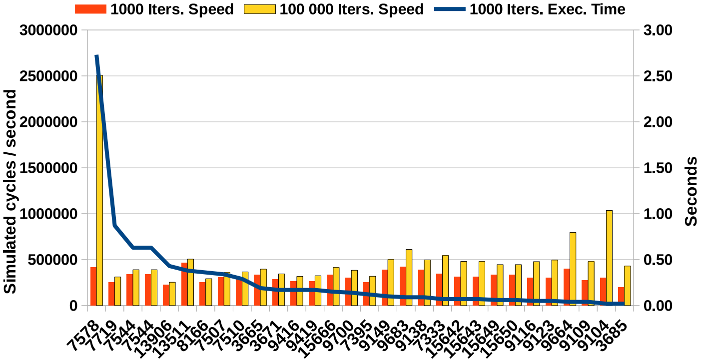
Speed on YALES2 Loops



Notes

- 281k cycles per second
- 0.13 second per loop
- 8 loops per second

Speed on AVBP Loops



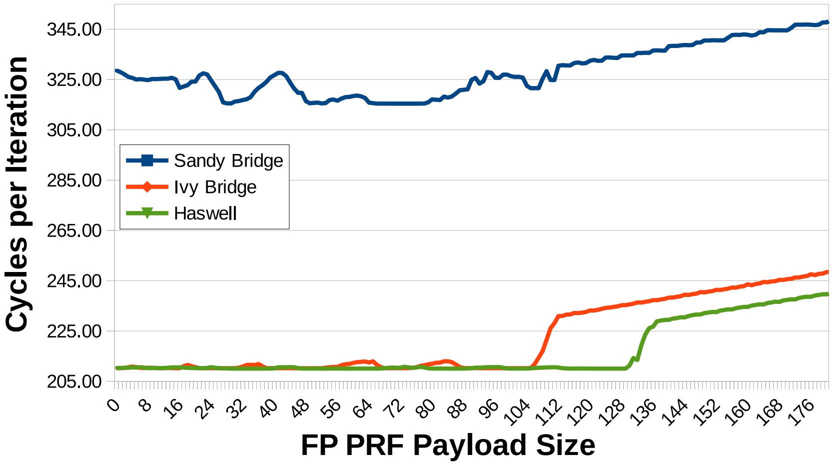
Notes

- 300k cycles per second
- 0.28 second per loop (0.19 w/o div outlier)
- 3.5 loops per second (5 w/o div outlier)

UFS: Adjusted Resource Sizes

Uarch	SNB	IVB	HSW
BB	48	48	48
LB	64	64	72
LM	32	32	32
FP PRF	112	113	138
Integer PRF	128	130	144
Overall PRF	141	165	177
ROB	165	168	192
RS	48	51	51
SB	36	36	42
ROB Microfusion	No	Yes	Yes
RS Microfusion	No	No	No

Resource Quantification (FP PRF)



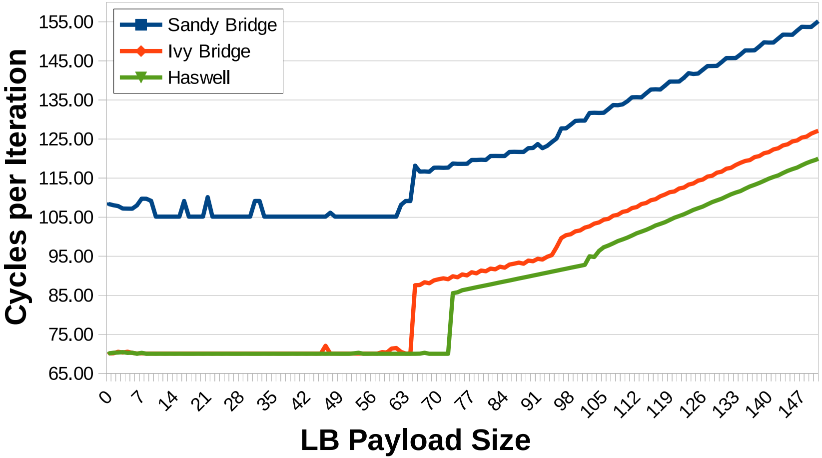
Resource Quantification (LB)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
2	VSQRTPD %xmm0, %xmm1	Jamming retirement
3	VSQRTPD %xmm0, %xmm1	Jamming retirement
4	VSQRTPD %xmm0, %xmm1	Jamming retirement
5	VSQRTPD %xmm0, %xmm1	Jamming retirement
6	VMOVUPS 0(%rsi), %xmm1	Payload
7	VMOVUPS 0(%rsi), %xmm1	Payload
8	SUB $1, %rdi	Loop Control
9	JG .LOOP	Loop Control

Notes

- Loads are dispatched in parallel with square roots (and leave the RS / LM)
- Example for *payload size = 2*

Resource Quantification (LB)



Resource Quantification (LM)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming dispatch
...	VSQRTPD %xmm0, %xmm1	Jamming dispatch
5	VSQRTPD %xmm0, %xmm1	Jamming dispatch
6	VMOVMSKPD %xmm1, %r13	Jamming dispatch
7	ADD %r13, %r12	Jamming dispatch
8	VMOVSS (%r12), %xmm2	Base Load
9	VMOVSS (%r12), %xmm2	Base Load
10	SUB $1, %rdi	Loop Control
11	JG .LOOP	Loop Control

Notes

- Loads are made to depend on *VSQRTPDs*
  - Cannot get dispatched in parallel with square roots from the same iteration (and stay in RS / LM)
- Example for *payload size = 2*

Resource Quantification (LM)