ONE VIEW: a tool for performance analysis agregation

W. Jalby

University of Versailles Saint Quentin en Yvelines/ECR

- Architectures, Applications and last but not least Tools also are becoming very complex
- A simple example: hardware performance counters:
  - Too many of them
  - Very little documentation if any. Broken counters are not publicized ☺
  - Needs detailed understanding of micro architecture to exploit them
  - Hard to distinguish between cause and consequence
  - When a resource is saturated, does it hurt performance ??
  - They change with every processor generation.
  - …..

MAIN LESSON: do not rely only on hardware events ☺

Code Developper Point of View

- Major issue: flat profile: hundred loops with each 1% coverage
- Strategy for optimization ??
- Bottleneck detection/analysis is not enough. KEY QUESTION: how much performance gain can be expected if bottleneck is removed or if a given optimization is applied
- Assessing ROI (Return on Investment) is essential
- Coverage information is not enough. Let us assume
  - Loop A: coverage 40% potential speedup (ROI) 1,1 x. Total application gain: around 4%
  - Loop B: coverage 20%, potential speedup (ROI) 2 x. Total application gain: around 10%

ONE VIEW GOAL/APPROACH

GOAL

- Guide the user through the optimization process
- Identify and assess optimization opportunities

ONE VIEW APPROACH

- Currently focusing on one core issues and more precisely compiler, vectorization and data locality
- Fully automated process: user provides binary and input data and just specifies analysis depth and gets MS Excel files with analysis results
- Simple summary with potential performance gain per loop basis and per optimization.
- Keep correlation between performance issues and source code
- Behind the scene, ONE VIEW will combine several tools for building its summary reports.

ONE VIEW ONE

- A single run
- Combines static analysis (CQA: Code Quality Analyzer) with simple sampling profiling
- Three static analysis targets
  - Impact of code cleaning
  - Impact of FP vectorization
  - Impact of full vectorization

CQA Performance model

- Works on binaries/asm code. Interfaces with MADRAS (our own disassembler/patcher) and INTEL XED
- Based essentially on instruction execution bandwidth. Latencies are used for analyzing dependence cycles.
- Uses a simple throughput model, takes into account ports structures, some FE limitations, BUT NOT OOO BACK END BUFFER SIZES
- In presence of branches, CQA identifies potential paths and performs path analysis
- Use of data access patterns (L, S, LS, LLS, etc ….) to model data access
- Repository of performance for all access patterns (filled by microbenchmarking)
- Four performance estimates: data in L1, L2, L3 and RAM

Pros and cons: due to its static nature

- ***Cons: accuracy***
- ***Pros: speed***

Two steps done at asm level

1. Suppress all of the non FP instructions
2. Run CQA on the modified code and compare CQA statistics with original

Useful for detecting some strange compiler behavior.

ATTENTION: notion of Clean here corresponds to HPC codes where FP instructions are the ones corresponding to Science ☺

Predicting Vectorization Impact (1)

Four steps done at asm level

1. Static analysis to detect reductions, strided access and memory patterns. Unknown stride values are assumed to be non unit
2. Unroll and jam ignoring potential dependencies due to memory access
3. Generation of a vector mockup ASM code:
   - use rule based decision to replace group of scalar instructions by equivalent vector instruction.
   - keep track (preserves) of register dependencies
   - Inserts if necessary data reshuffling instruction to combine scalar values into the same vector register
4. Run vector mockup through CQA to get performance estimates

Vector Mockup code is automatically generated but not executable (address computation not correctly handled)

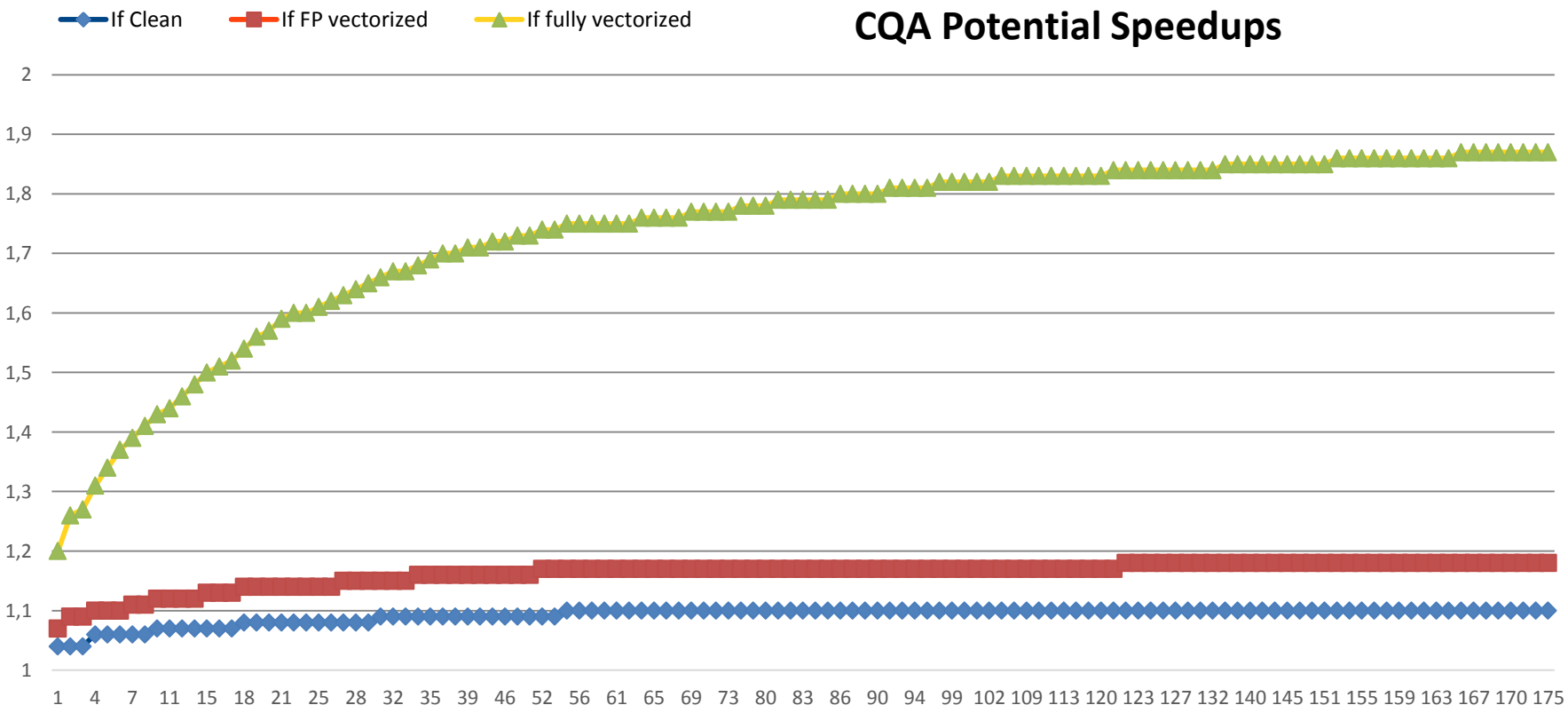
Predicting Vectorization Impact (2)

A few refinements

1. Generate partial vectorization estimates (FP only) or full vectorization estimates (selected at the mockup generation level)
2. Generate variants depending upon aligned versus non aligned data access
3. Instead of using standard scalar ASM code, use compiler to generate better scalar asm code through directives.

ONE-View standard coverage order

- Results showing the potential speedup for a simple vectorization L1 model
- Loops ordered by coverage
- Yales 2 (CFD/Combustion application)

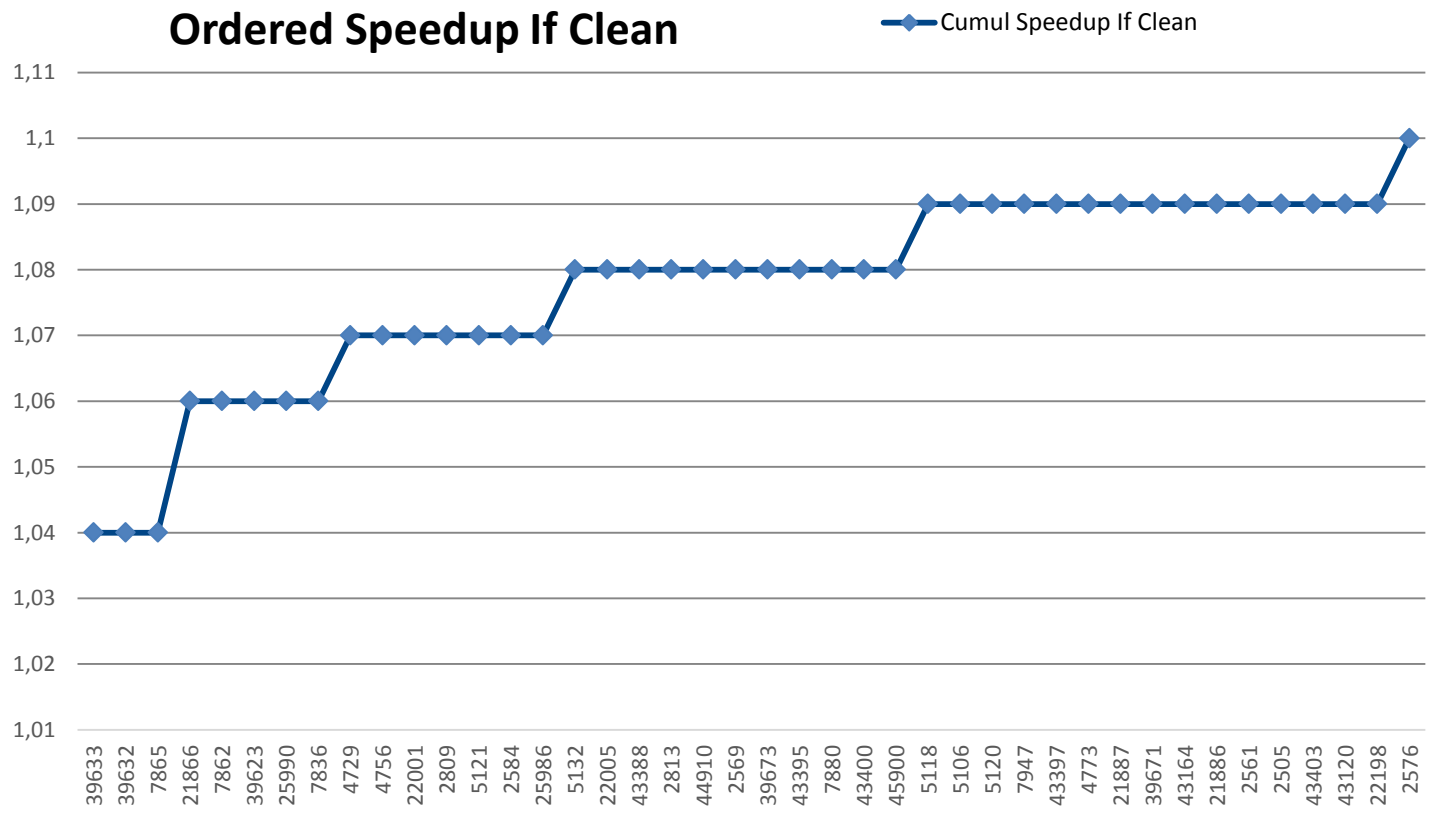


ONE VIEW: ROI

ID	Coverage (% app. time)	CQA speedup if clean	CQA speedup if FP arith vectorized	CQA speedup if fully vectorized	CQA speedup if no inter-iteration dependency	CQA speedup if next bottleneck killed
Loop 39633	**22,02**					
Path 1		1.23	1.45	3.94	1.00	1.03
Loop 39632	**5,87**					
Path 1		1.00	1.31	3.82	1.00	1.20
Loop 7865	**4,51**					
Path 1		1.02	1.00	1.21	1.00	3.65
Loop 21866	**2,83**					
Path 1		1.65	1.54	4.00	1.00	1.42
Loop 7862	**2,5**					
Path 1		1.07	1.07	4.00	1.00	1.05
Loop 39623	**2,01**					
Path 1		1.03	1.04	4.00	1.00	1.13
Loop 25990	**1,25**					
Path 1		1.39	1.69	4.00	1.00	1.13
Loop 7836	**1,2**					
Path 1		1.09	1.06	3.49	1.00	1.09
Loop 4729	**1,2**					
Path 1		1.57	1.60	5.68	1.00	1.33
Path 2		1.73	1.65	5.85	1.00	1.36
Path 3		1.73	1.65	5.85	1.00	1.36
Path 4		2.00	1.73	6.10	1.00	1.39

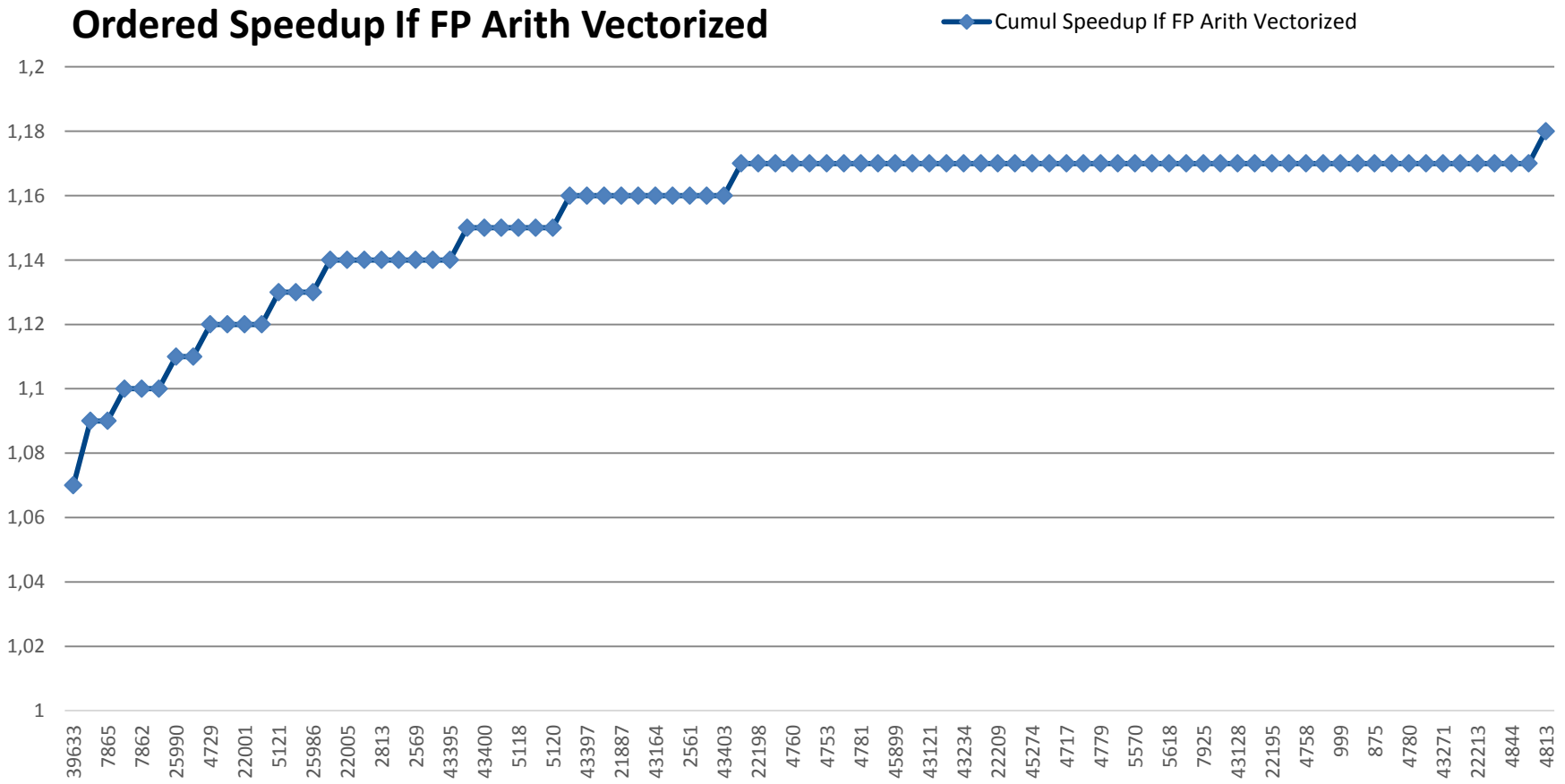
ONE VIEW: ROI order, clean code

- Results showing the potential speedup if “clean code” generated by the compiler, for the YALES 2 application (*MS Flame* model)
- Loops ordered by potential performance gain (not coverage).



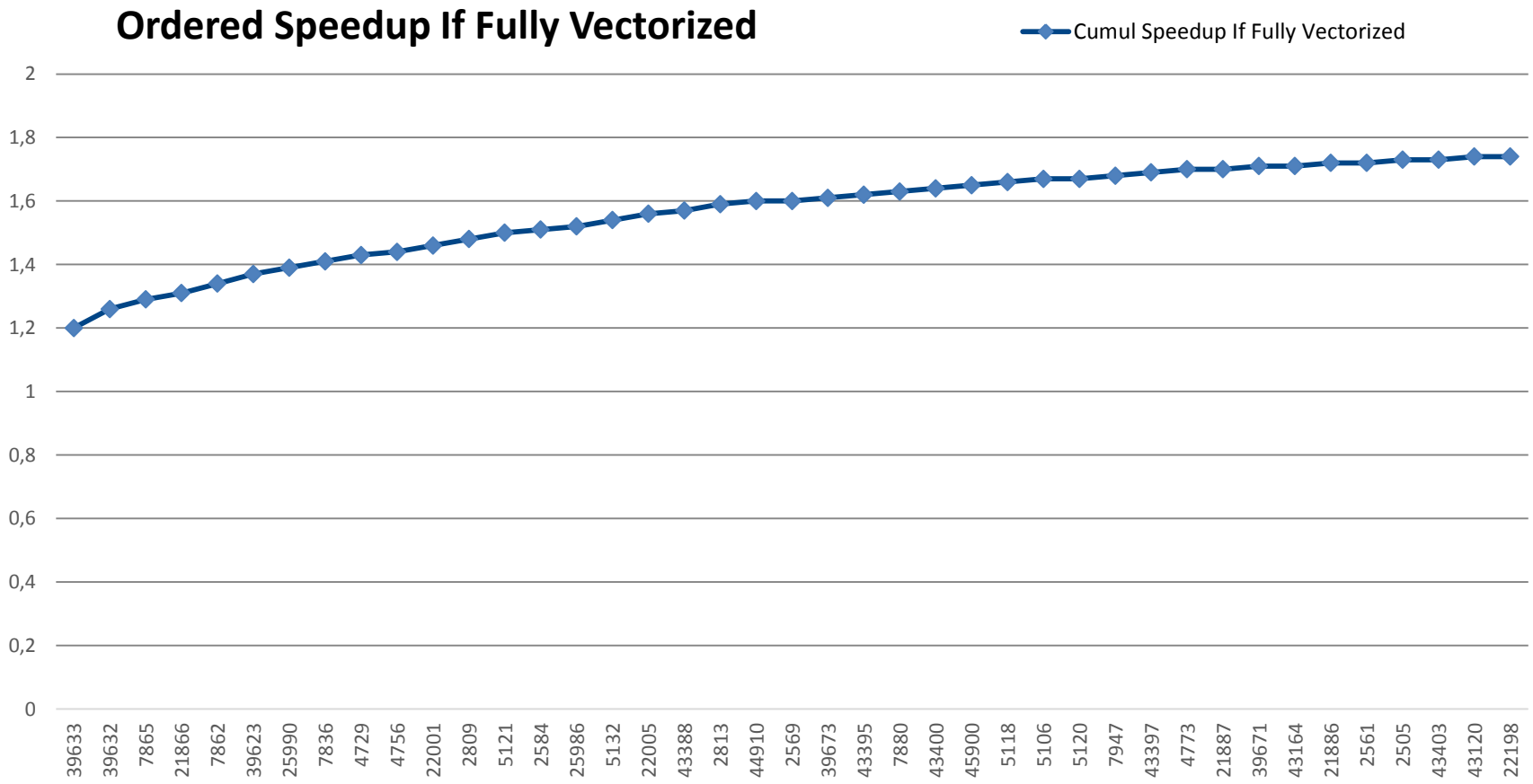
ONE VIEW: ROI order, FP Arithm vectorized

- Results showing the potential speedup if FP arith vectorized, for the YALES 2 application (*MS Flame* model).
- Loops ordered by potential performance gain (not coverage).



ONE VIEW: ROI order, fully vectorized

- Results showing the potential speedup if fully vectorized, for the YALES 2 application (*MS Flame* model)
- Loops ordered by potential performance gain (not coverage).

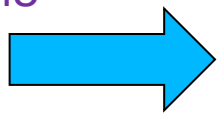


ONE VIEW "TWO"

- Three to five runs
- Focus on Dynamic Analysis
  - Full tracing analysis: identifies different behavior depending upon call site
  - Iteration counts distribution, stride values.
  - Selection of statistically representative sets of loop instances
  - Warns user about low quality measurements (durations less than 500 cycles)
- Data locality behavior analysis
  - Aims at identifying potential performance gain due to perfect blocking
  - Uses DECAN (Differential Analysis)
- Combines static analysis (CQA: Code Quality Analyzer) with simple sampling profiling

Original ASM

```
VMOVSD (%RDX,%R15,8), %YMM4
VMOVSD (%RDX,%R15,0), %YMM5
VADDSD YMM4, YMM5, YMM6
VMOVSD YMM6, (%RAX,%R15,4)
INC %R15
CMP %R15, %R12
JB
```



Modified ASM

```
VMOVSD 0x17f45(%RIP), %YMM4
VMOVSD 0x17f85(%RIP), %YMM5
VADDSD YMM4, YMM5, YMM6
VMOVSD YMM6, (%RAX,%R15,4)
INC %R15
CMP %R15, %R12
JB
```

Mem1, 2, 3 standard memory address, in general moving across address space

RIP Based address invariant across iterations: initial L1 miss than on subsequent iterations L1 hits

Issues/Extensions

- Modified code is a priori incorrect. Two issues:
  - Make sure that the modified asm does not modify critical data
  - Restore the correct memory state. After executing modified ASM, execute standard ASM
- Measure and compare performance of modified ASM versus original ASM
- Modified ASM corresponds to all operands accessed from L1
- Modified ASM executed in the same context as original

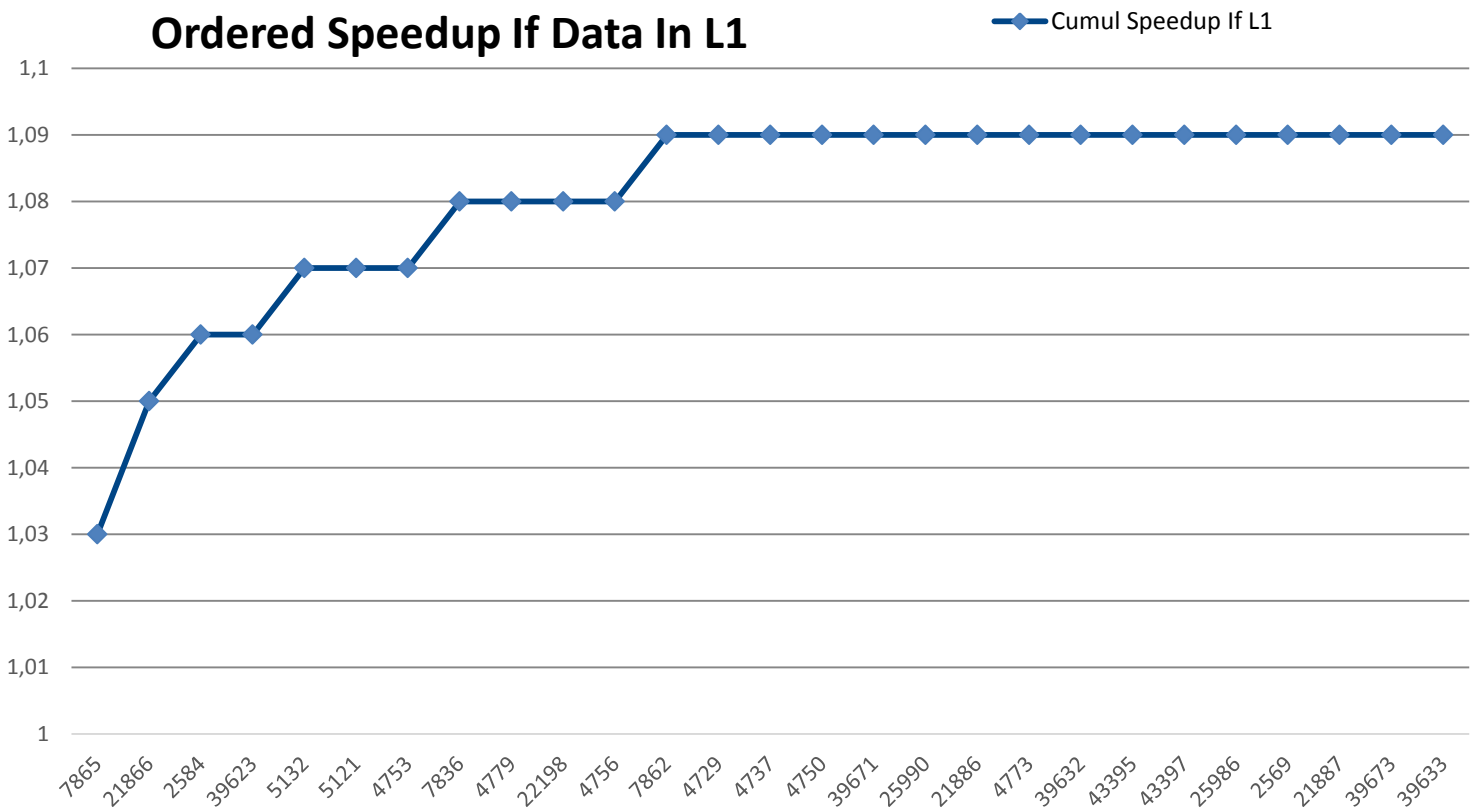
Perfect analysis of potential gain due to L1 access

Extensions:

- Can be applied selectively to one load or a few loads: analysis of data restructuring
- Can be applied to other instructions, more possible transformations: ONE VIEW "THREE"
- Straightforward extensions to OpenMp/ MPI programs

ONE VIEW ROI DL1

- Results showing the potential speedup if all data was in L1 cache for the YALES 2 application (*MS Flame* model)
- Loops ordered by potential performance gain (not coverage).



Work in Progress/Extensions

- Replace CQA by a simplified CPU OoO simulator to take into account limited buffer sizes
- Upgrade vector modeling to take into account dynamic memory system behavior.
- Analysis of L2/L3 blocking impact
- Analyze impact of latency / bandwidth of different cache levels
- Analysis of improved prefetching
- Analysis of improved operand location (MCDRAM, local versus remote).