Performance Analysis and Optimization Challenges

Zakaria Bendifallh, William Jalby, José Noudohouenou, Emmanuel Oseret, Vincent Palomares, Andres Charif-Rubial

UVSQ – Perfcloud, Exascale Computing Research

Outline

- Introduction
- Motivation
- Framework
- Dealing with Real Applications
- Conclusions and Future Works

Introduction: *context of performance analysis*

Hardware architectures are becoming increasingly complex

- Complex CPU: out of order, vector instructions
- Complex memory systems: multiple levels including NUMA, prefetch mechanisms
- Multicore introduces new specific problems, shared/private caches, contention, coherency
- Software stack is also becoming more and more complex

Each of these hardware mechanisms introduce performance improvement but to work properly, they require specific code properties

Performance pathologies: situations potentially inducing performance loss: hardware poor utilization

Individual performance pathologies are numerous but finite

Introduction: *usual performance pathologies (1)*

Pathologies	Issues	Work-around
ADD/MUL balance	ADD/MUL parallel execution (of FMA) underused	Loop fusion, code rewriting e.g. Use distributivity
Non pipelined execution units	Presence of non pipelined instructions: DIV, SQRT	Loop hoisting, rewriting code to use other instructions eg. x86: div and sqrt
Vectorization	Unvectorized loop	Use another compiler, check option driving vectorization, use pragmas to help compiler, manual source rewriting
Complex CFG in innermost loops	Prevents vectorization	Loop hoisting or code specialization

Introduction: *usual performance pathologies (2)*

Pathologies	Issues	Work-around
Unaligned memory access	Presence of vector-unaligned load/store instructions	Data padding, use pragma and/or attributes to force the compiler
Bad spatial locality and/or non stride 1	Loss of bandwidth and cache space	Rearrange data structures or loop interchange
Bad temporal locality	Loss of perf. due to avoidable capacity misses	Loop blocking or data restructuring
4K aliasing	Unneeded serialization of memory accesses	Adding offset during allocation, data padding
Associativity conflict	Loss of performance due to avoidable conflict misses	Loop distribution, rearrange data structures

Introduction: *usual performance pathologies (3)*

Pathologies	Issues	Work-around
False sharing	Loss of BW due to coherence traffic and higher latency access	Data padding or rearrange data structures
Cache leaking	Loss of BW and cache space due to poor physical-virtual mapping	Use bigger pages, blocking
Load unbalance	Loss of parallel perf. due to waiting nodes	Balance work among threads or remove unnecessary lock
Bad affinity	Loss of parallel perf. due to conflict for shared resources	Use numactl to pin threads on physical CPUs

Introduction: *Analysis of current tool set 1/3*

- Lack of global and accurate view: no indication of performance loss (or alternatively ROI)
  - Performance pathologies in general but no hint provided on performance impact (cf VTUNE with performance events): we do not know the pay off if a given pathology is corrected
  - Worse, the lack of global view can lead you to useless optimization: for example, for a loop nest exhibiting a high miss rate combined with div/sqrt operations, it might be useless to fix the miss rate if the dominating bottleneck is FP operations.
  - Source code correlation is not very accurate: for example with VTUNE relying on sampling, some correlation might be exhibited but it is subject to sampling quality and out of order behavior.

Introduction: *Analysis of current tool set 2/3*

- Very often, most of the tools rely on a single technique/approach (simplified view but globally correct)
  - Vtune is heavily relying on sampling and hardware events
  - Scalasca/Vampir/Tau is heavily relying on tracing and source code probe insertion
  - Sampling aggregates everything together (all instances): might be counterproductive
  - In practice, flexibility has to be offered: tracing might be more efficient than sampling and vice versa.

Introduction: *Analysis of current tool set 3/3*

- A common pitfall: “Hardware events/counters can explain everything” ☺
  - Counting the number of cache misses is useless unless you know the average cost of a cache miss which can vary from 5 cycles (near hit) up to 200 cycles (ineffective prefetching, access all the way to DRAM)
  - High values for counters: Reservation Station (ROB) buffers full. Is it good or bad ?? It just shows that input rate is larger than output rate. Everything depends upon these rate values.
  - Counters hard to correlate with source code
  - Counters change from one processor generation to the next: not only names but sometimes semantics.

Introduction: *(usual performance pathologies)*

- Well known
- But:
  - How to find them ?
  - How much do they cost ?
  - What to do when multiple pathologies are present ?
- Need to quantify/hierarchize them

Our Objectives: *Techniques & Modeling*

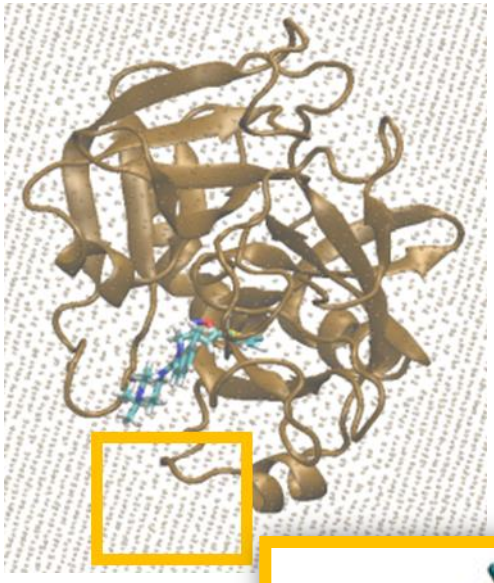
- Get a global hierarchical view of performance pathologies/bottleneck
- Estimate the performance impact of a given performance pathology while taking into account all of the other pathologies present
- Use different tools for pathology detection and pathology analysis
- Perform a hierarchical exploration of bottlenecks: the more precise but expensive tools are only used on a specific well chosen cases

Motivation: Example (1)

POLARIS(MD) Loop

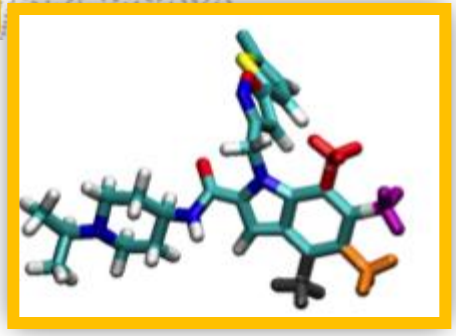
- Molecular Dynamics
  - Based on the Newton equation: $m\vec{a} = -\vec{\nabla} U_{pot}$
  - Multiscale
- Developed at CEA (French energy agency) by Michel Masella
- 60K LOC, Fortran 90
- OMP, MPI, OMP+MPI

Example of multi scale problem: Factor Xa, involved in thrombosis



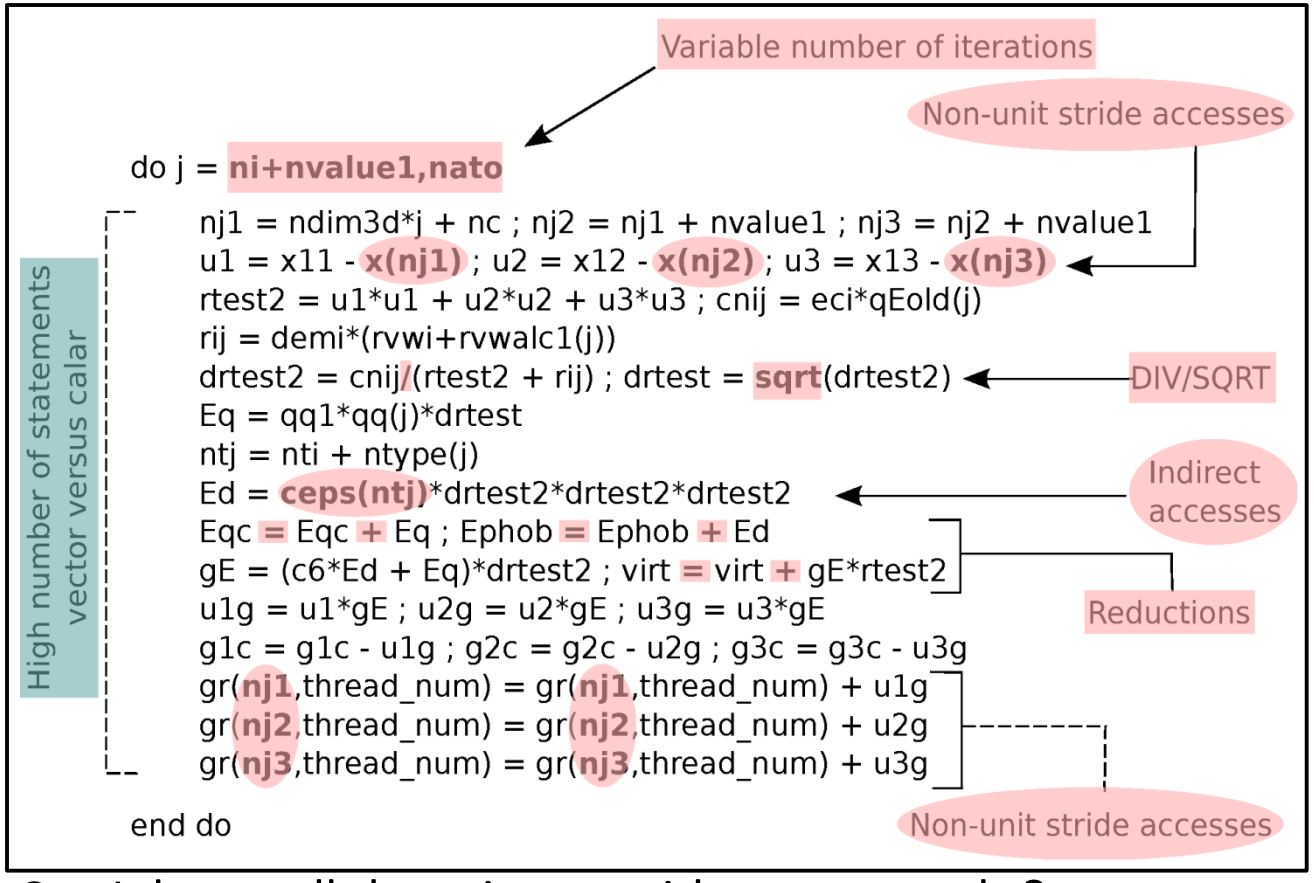
$(7.46\ nm)^3$

Anti-Coagulant



Motivation: Example (2)

Source code and issues



1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Vector vs scalar

Special issues:
Low trip count: from 2 to 2186 at binary level

Can I detect all these issues with current tools ?

Can I know potential speedup by optimizing them ?

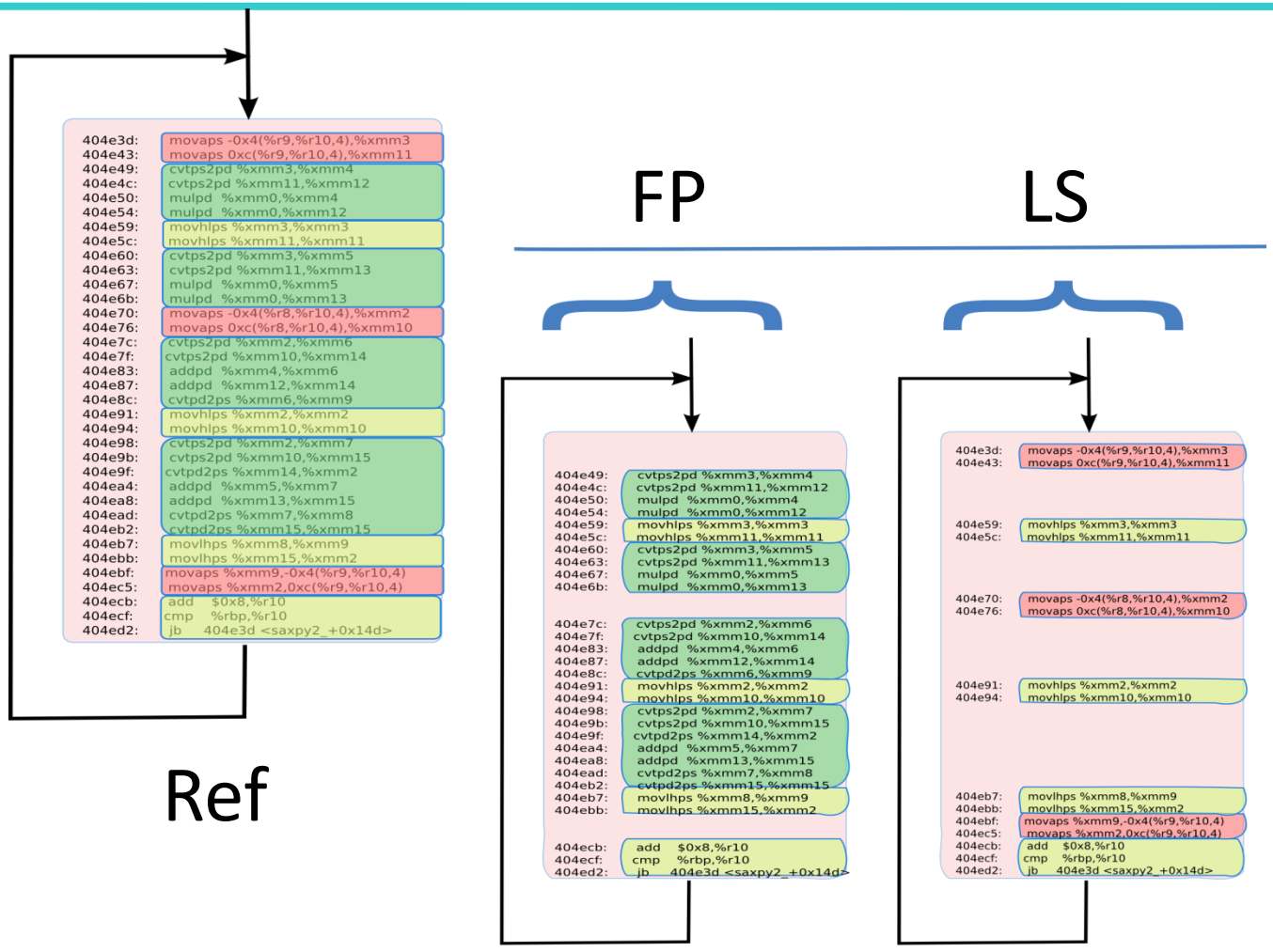
Tools: *CQA*

- CQA = Code Quality Analyzer
- Objectives (provides):
  - Best performance estimation (assuming data in L1)
  - Code quality information (and optimization hints for compiler flags and source transformations)
  - First estimation of bottlenecks hierarchy
- Statically analyzes innermost loops binaries: builds DDG
- Supports Intel 64 micro-architectures from Core 2 to Ivy Bridge (Haswell coming soon)
- Provides metrics and reports at both low and high abstraction levels

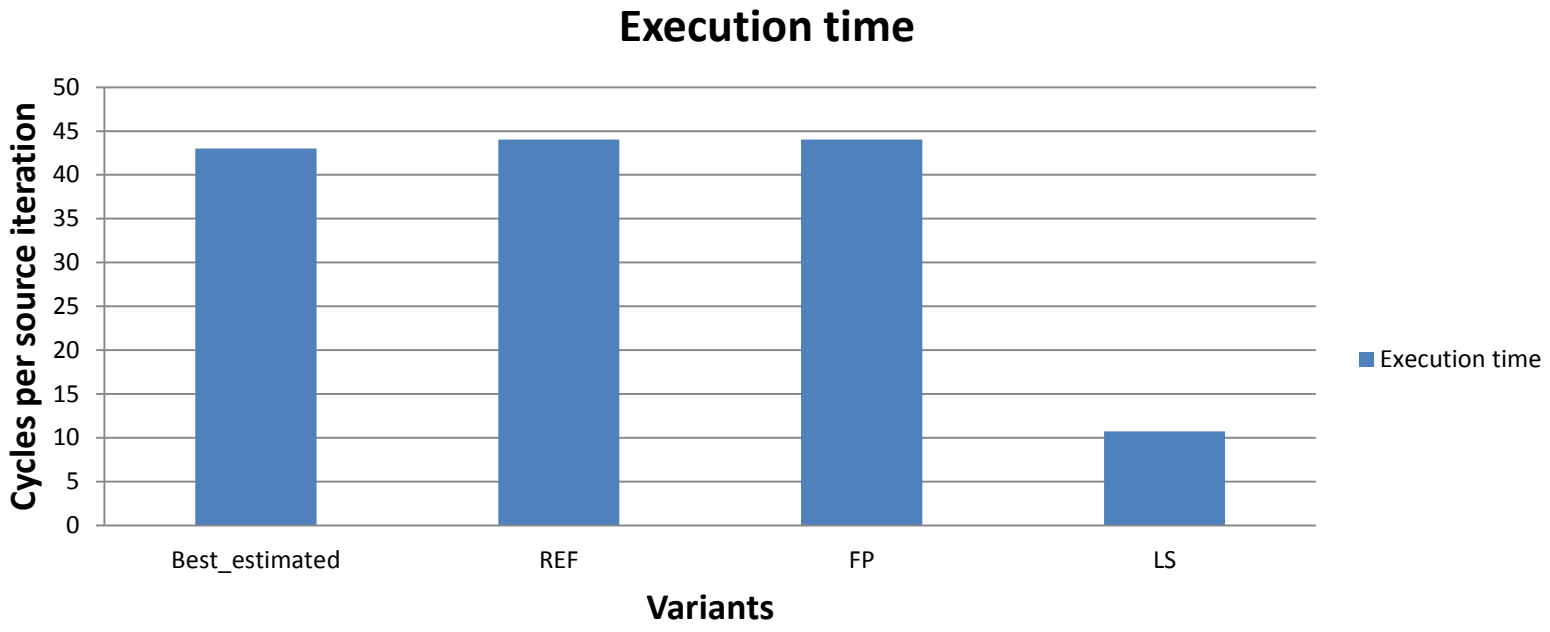
Tools: *Differential Analysis*

- A key technique in our approach: Differential Analysis (DECAN):
- Original Binary (B): I1, I2, I3, I4 (let us assume I2 is a load Instruction accessing an unknown cache level)
- Patched Binary (B'): I1, I'2, I3, I4 (I2 has been replaced by I'2: forcing L1 access)
- Perf(B) – Perf(B') = Marginal Cost of original I2 data access
- Differential analysis allows:
  1. To detect/isolate costly sequence of instructions
  2. To estimate accurately their impact on global performance

Tools: *Diff. Analysis (2/3, transformations)*

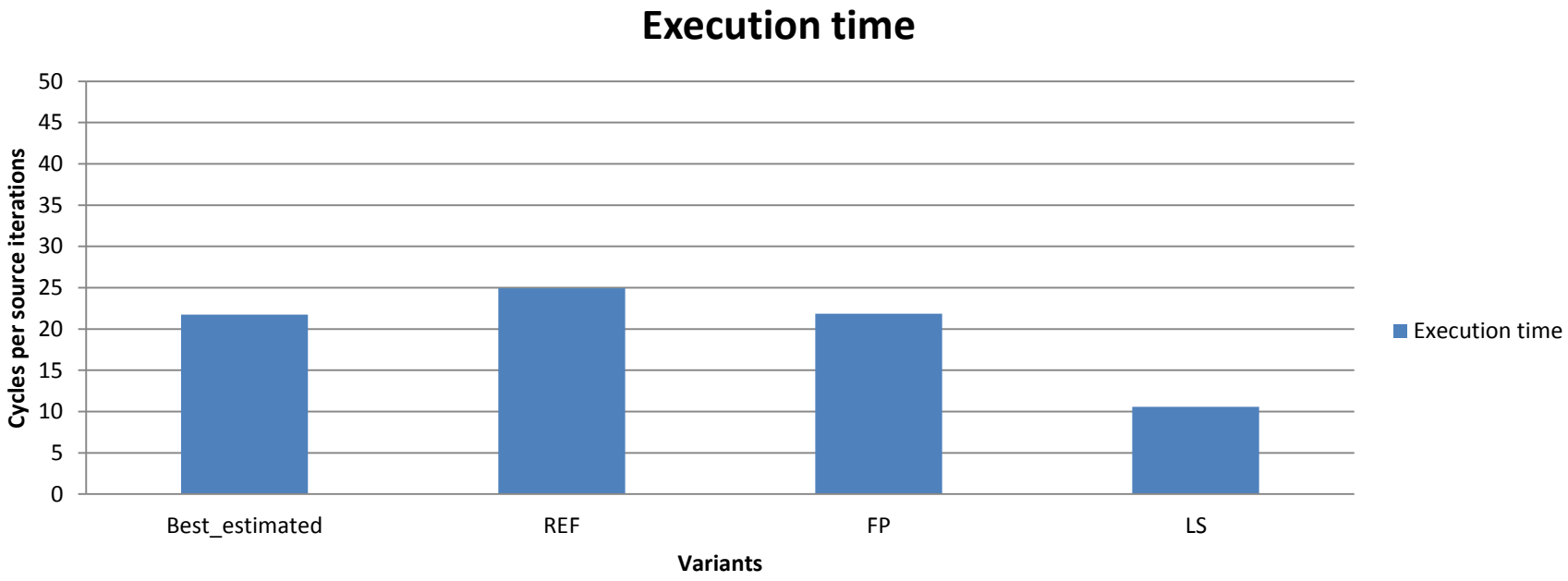


Case study: *Original code: Dynamic properties (2)*



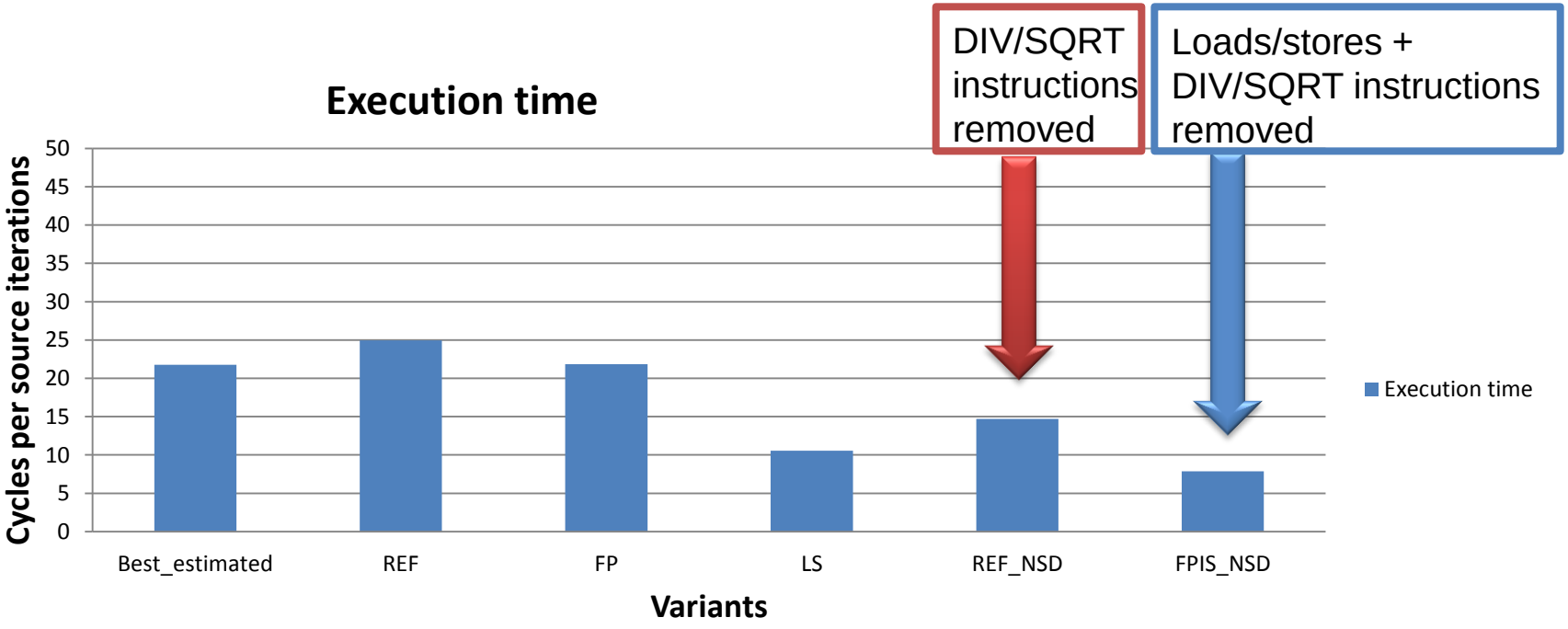
Best Estimated: CQA results
REF: Original code FP: only FP operations are kept LS only Load Store instructions are kept.
FP / LS = 4,1: FP is by far the major bottleneck

Case study: *Dynamic properties after vectorization: using SIMD directive*



FP / LS = 2,07 - Initial value was at 4,1

Case study: *one step further*



REF_NSD : removing DIV/SQRT instructions provides a 2x speedup
=> the bottleneck is the presence of these DIV/SQRT instructions
FPLS_NSD : removing loads/stores after DIV/SQRT provides a small additional speedup
Conclusion: No room left for improvement here (algorithm bound)

Tools

- MAQAO (Modular Assembly Quality Analyzer and Optimizer)
  - Operates on ELF64 binary files (Linux x86_64 supported)
  - Static (CQA) and dynamic (Diff. Anal. + MTL) analysis
- Differential Analysis
  - Uses MAQAO (for disassembling, patching…)
  - Measures performance impact of some instructions in loop bodies

Tools: *MicroTools: Microbenchmarking*

- An automatic solution for microbenchmarks generation in an easy and reusable way
- Measuring the machine's workload per piece of code
- Useful for getting an idea of potential performance impact of different pathologies.

Methodology
(basic benchmarking)

- Evaluate ideal bandwidth for load instructions
- Best monocore results among various stream experiments
  - With or without prefetch instructions
  - With or without splitting streams

Instruction	L1	L2	L3	RAM
movaps	30,91	18,20	10,73	5,41
movups	28,69	17,08	10,73	5,32
movsd	15,72	11,56	10,28	5,59
movss	7,93	6,83	6,33	4,96

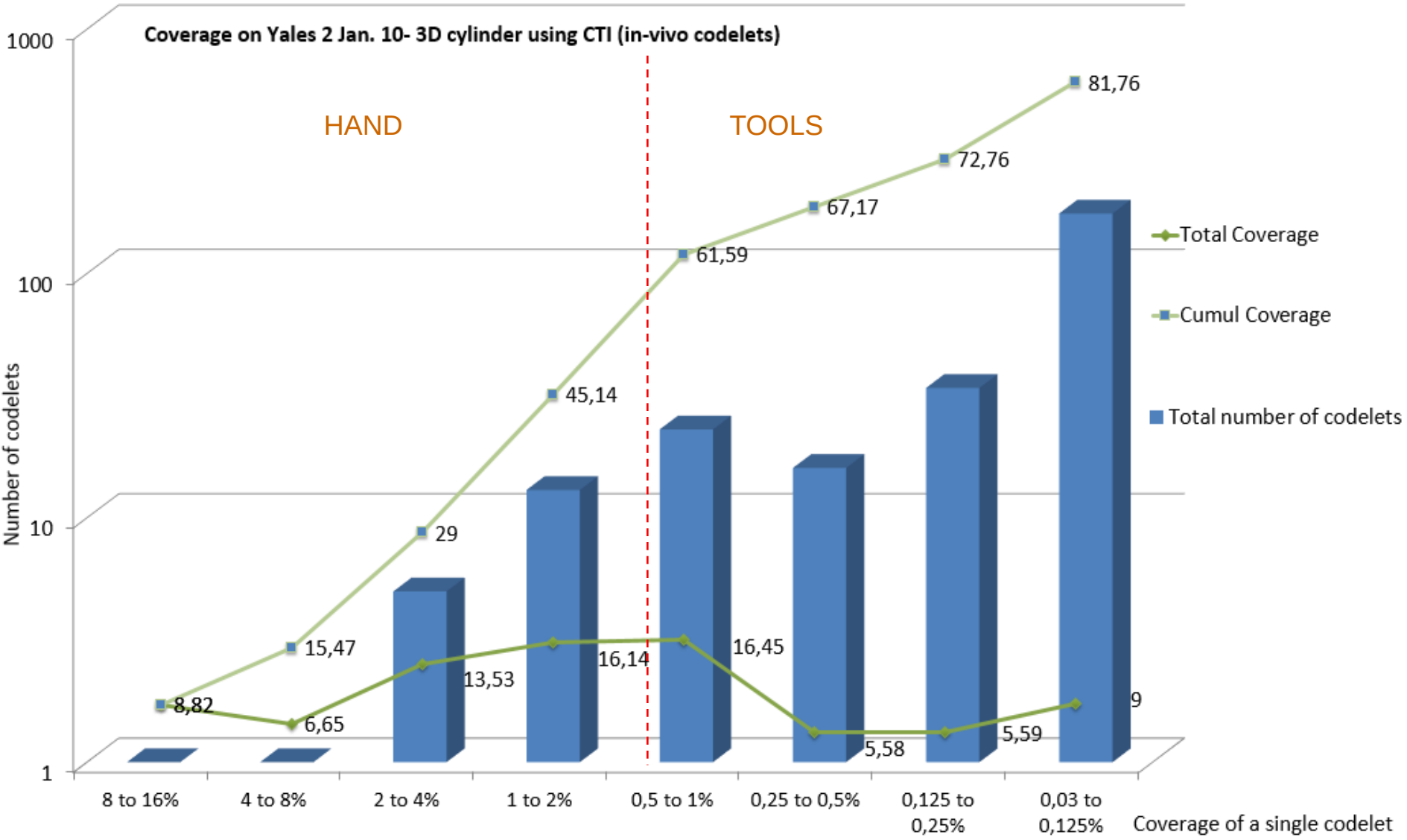
~=

Bytes per Cycle

Methodology
(Main steps)

1) Basic bandwidth benchmarks (once per architecture)
2) Keep loops up to 80% execution time (sampling)
3) Trace iterations count for each instance
4) Check for short loop trip count
5) Run differential analysis to qualify/quantify bottlenecks (assess if CPU or memory bound…)
6) Run static analysis
7) Investigate CPU or memory bound issues

Real Application Behavior: MAQAO Perf on Yales2 (CORIA)



Analysis on Yales 2 (1)

5 – Clustering Approach

- First step
  - Ongoing work
  - 10 static metrics provided by CQA
  - Innermost
- Yales 2 3D cylinder 0110: 20 clusters
- Cluster A (12 codelets): AXPY

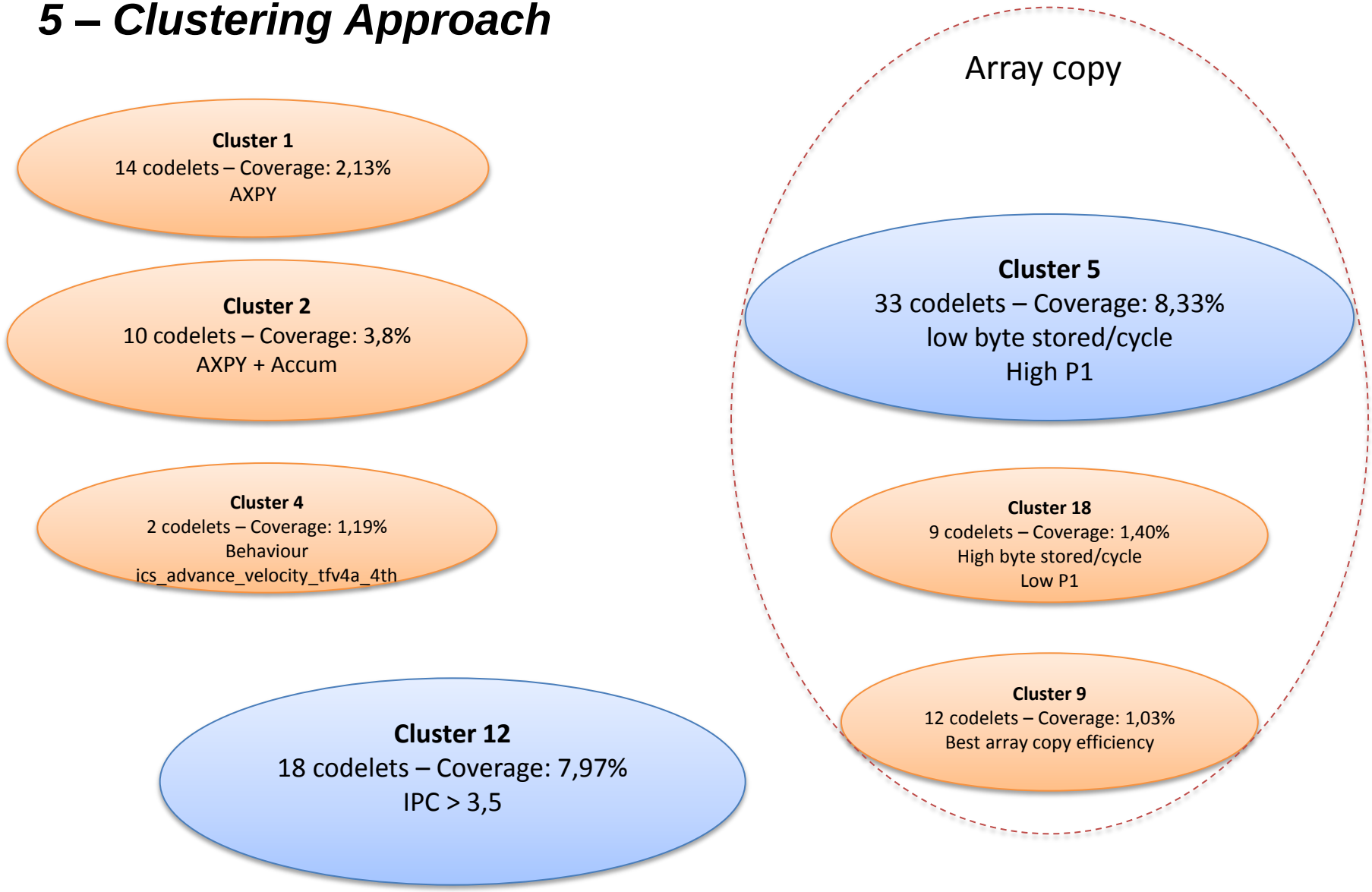
```
grad_ptr%int_comm_r3%val(1:grid%ndim,j,inoi) =          grad_ptr%int_comm_r3%val(1:grid%ndim,j,inoi)  +

                scal_val_r1%val(j)*normal(1:grid%ndim)
```

- Cluster B (29 codelets): AXPY and accumulation
- Cluster C (15 codelets): Array copy
- Cluster D (8 codelets): Codelets with good vectorization
- Other clusters…

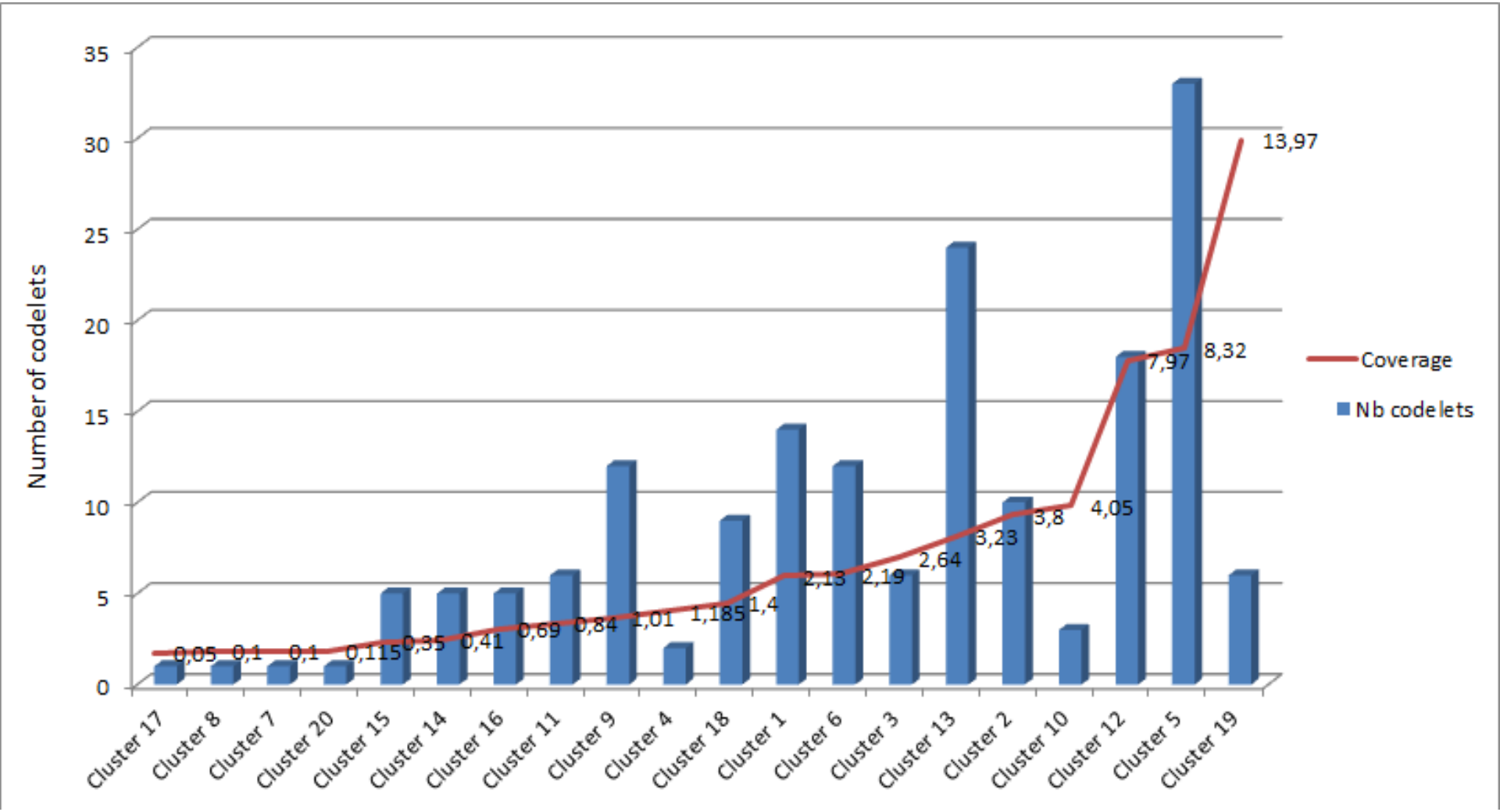
Analysis on Yales 2 (2)

5 – Clustering Approach



Analysis on Yales 2 (3)

5 – Clustering Approach

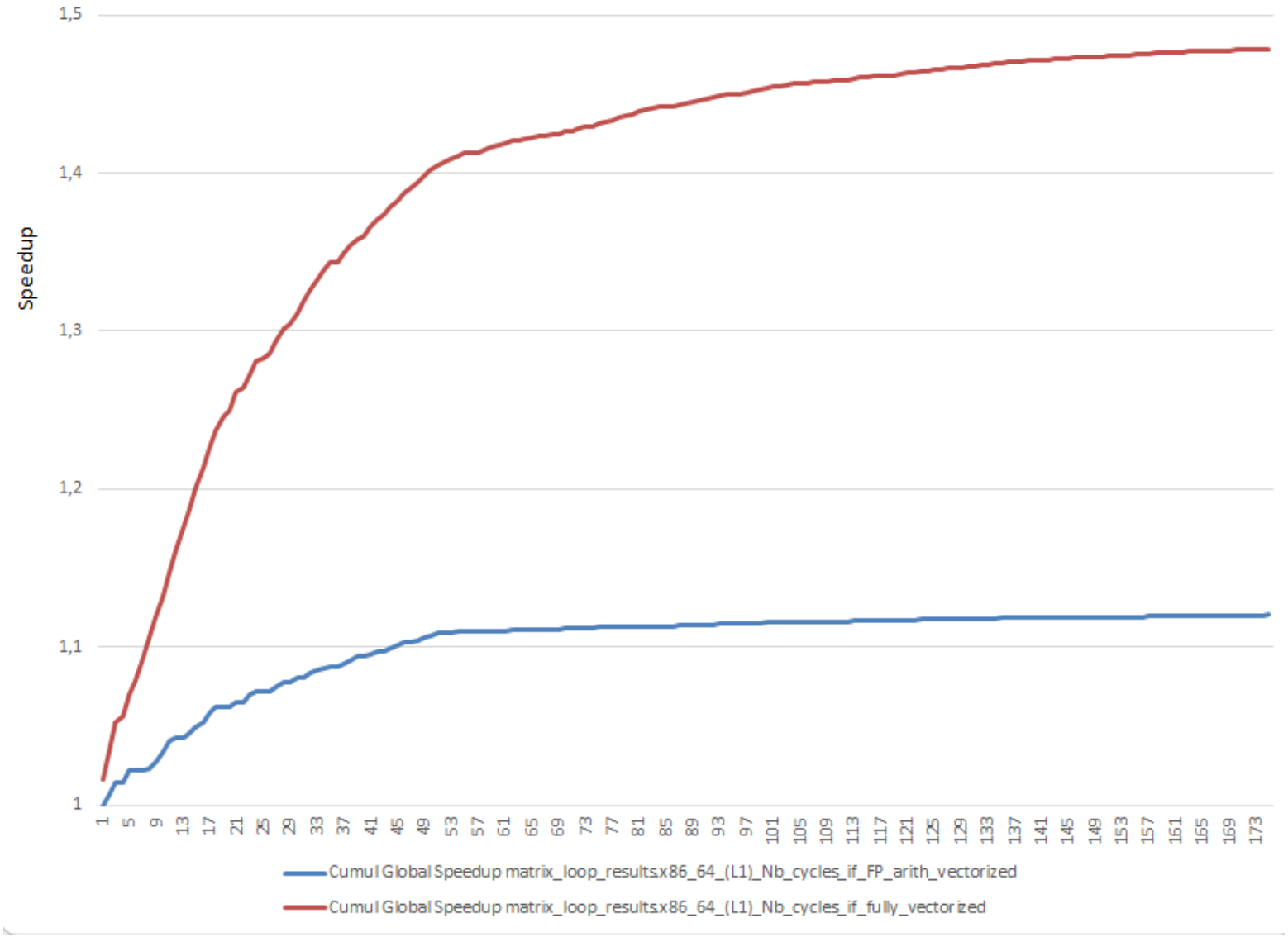


Vectorization Analysis

Vectorization: 2 levels

- Fully vectorized
  - All the uops/instructions are considered as vectorized
- FP arith
  - Uops related to float computation only are considered as vectorized (no load / store, no address computation)

2- Major Results: Potential Impact of Vectorization on Yales2 MS 1D Flame



Conclusions and future works

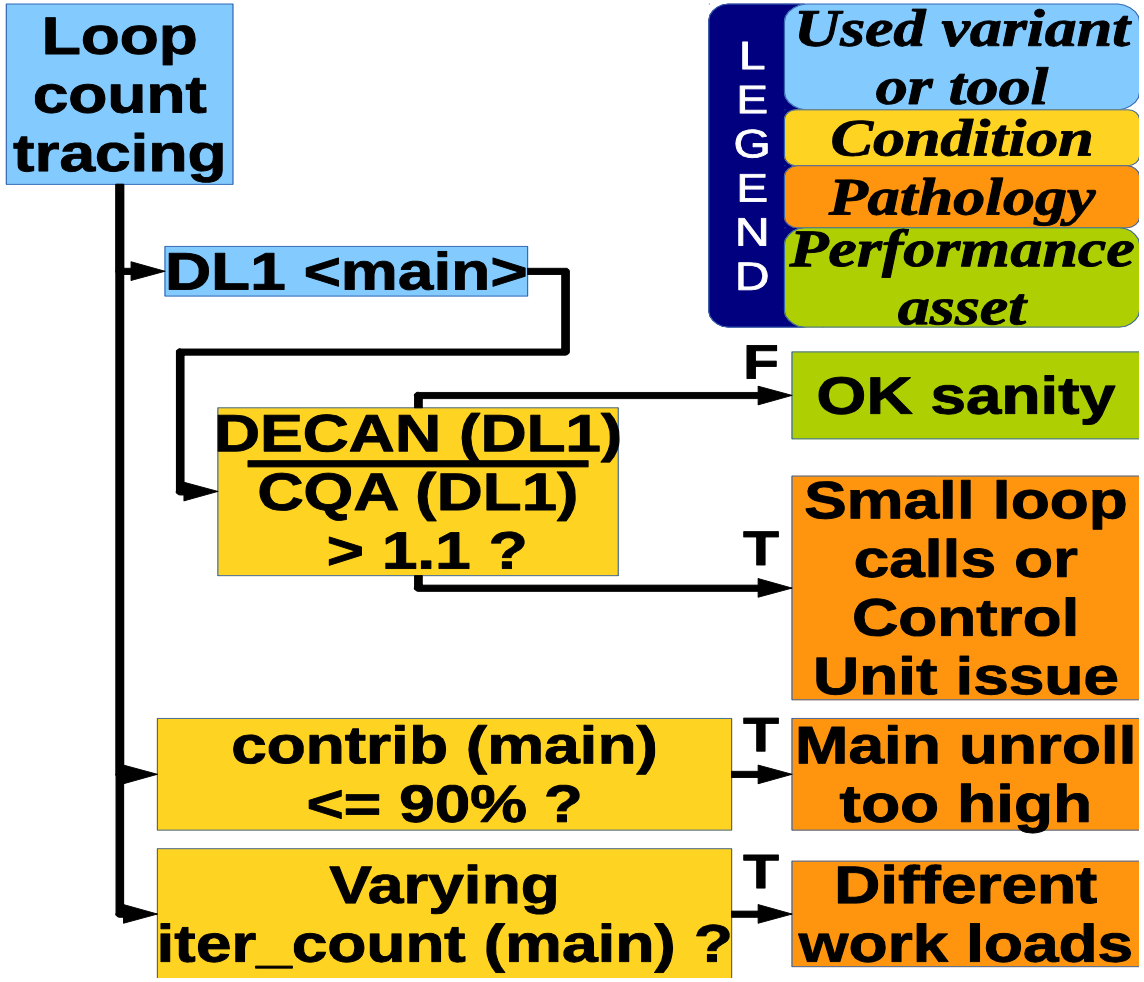
- Differential analysis provides first-order bottlenecks and related ROI
- For CPU-bound loops, MAQAO CQA provides finer, per-issue ROI
- Real Applications requires more sophisticated techniques to handle large number of loops
- Performance analysis should cover code design choices for upcoming architectures
- Future works
  - Refine memory bottleneck analysis
  - Fully automate the methodology
  - Scalability bottlenecks (OpenMP, MPI) ?
  - Cost notion: weighting ROI by code complexity

Methodology
(hotspot identification)

- Identify hot loops using cycles sampling:
  - collect the set of hot loops which represent 80% of execution time
- Identify the most significant loop calls using Loop Count Tracing
  - Constructs the deciles from the traces
  - Select a candidate loop call from each decile
- Determine the global ROI of the loops
  - launch the LSIS and FPIS versions for the selected loop calls
  - Calculate for each loop:
    - ROIi = max(LSISi,FPISi)/min(LSISi,FPISi) , then:
    - G_ROIi = ROIi ∗ EXEC TIMEi
- Order the loops list following the biggest score of G_ROI

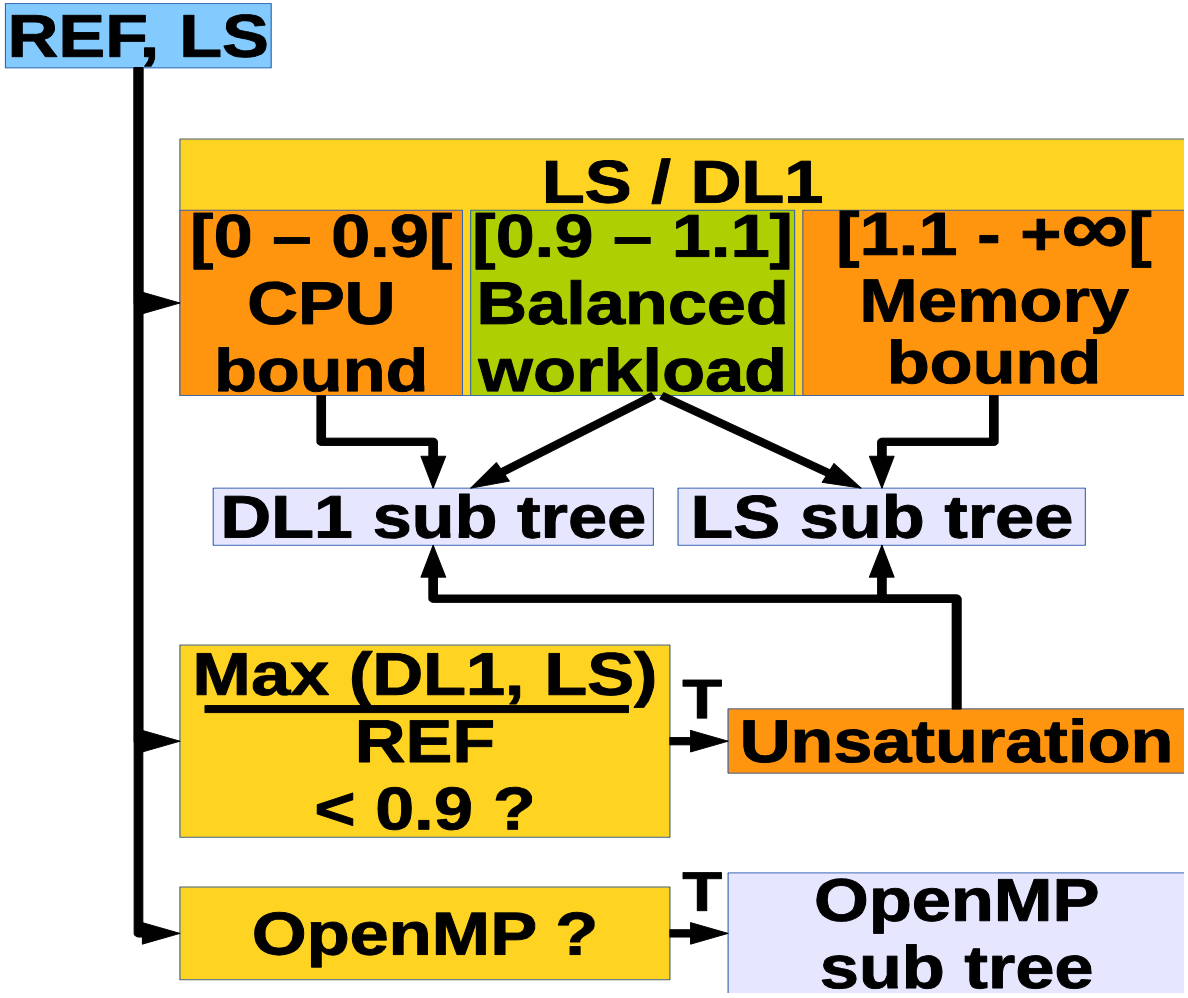
Methodology

(sanity tree)

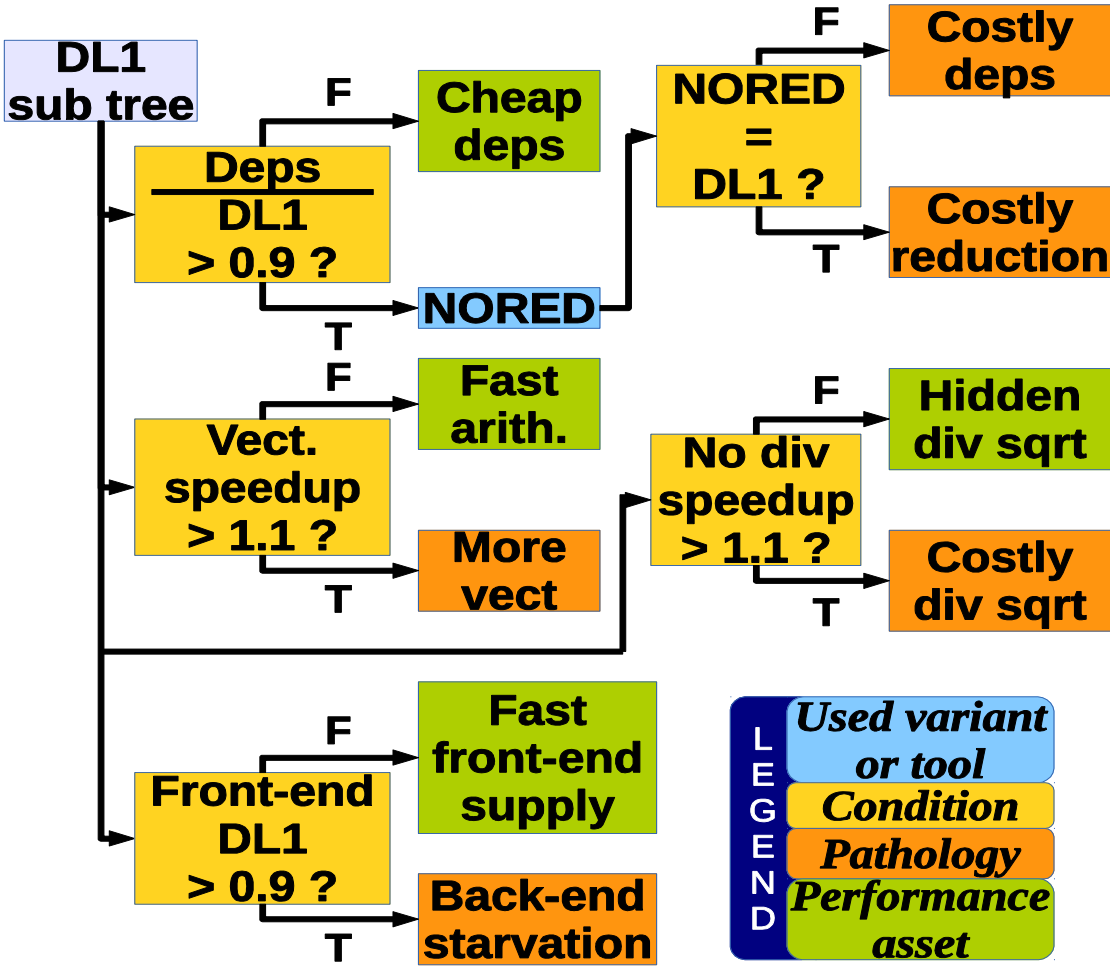


Methodology

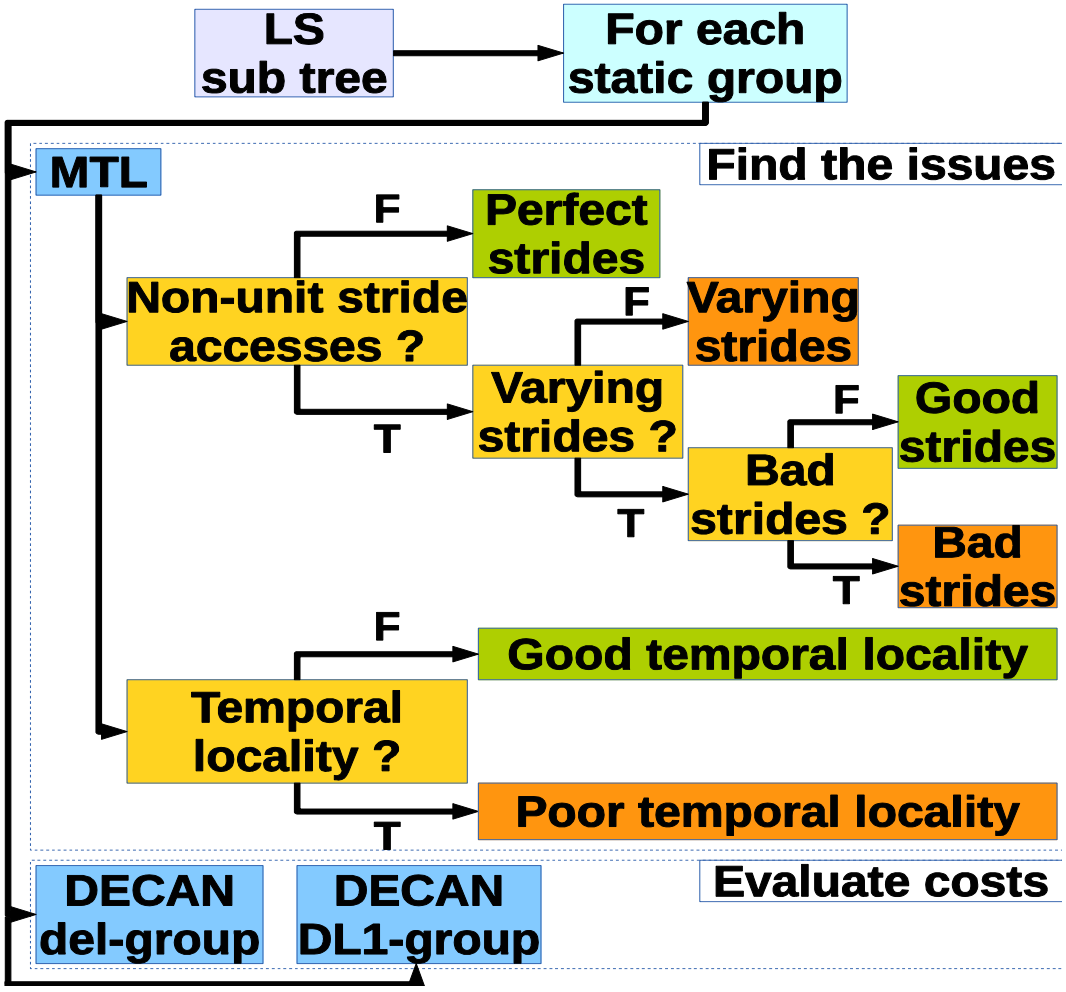
(main tree)



Methodology
(CPU bound tree)

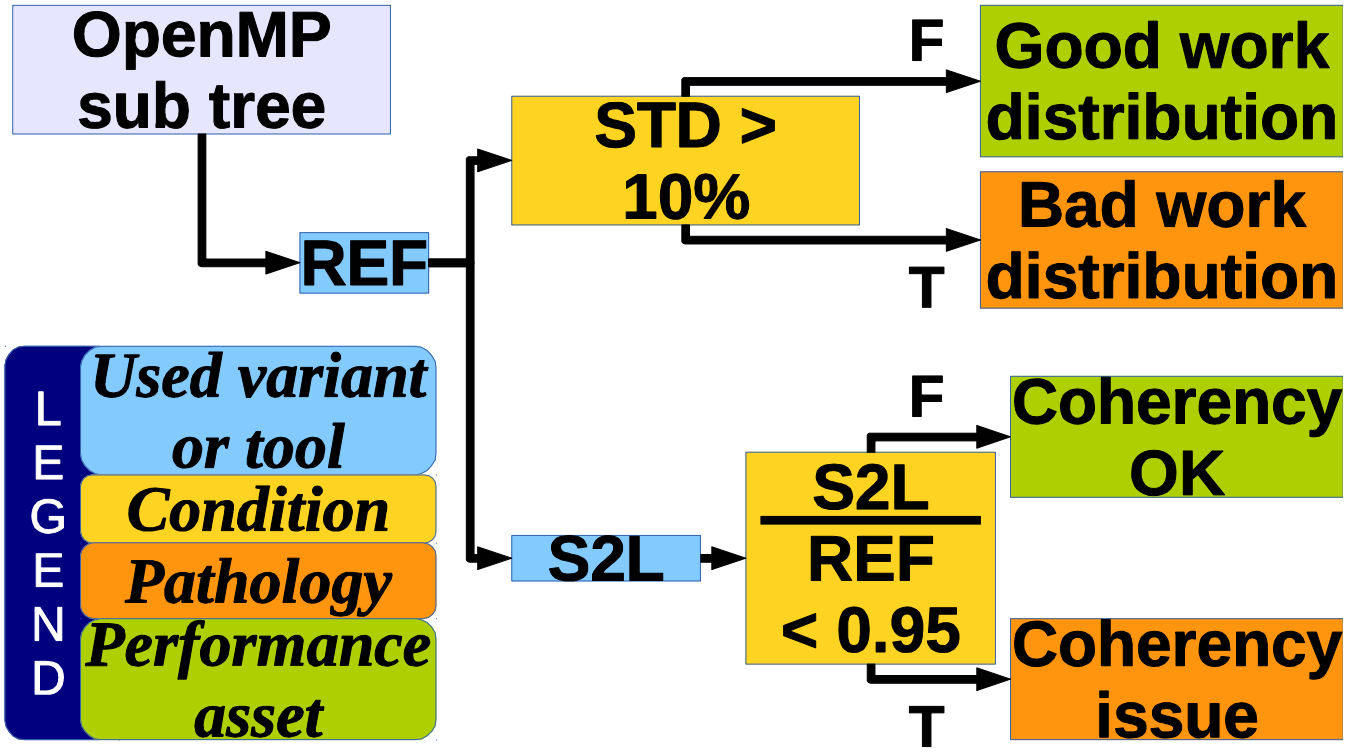


Methodology
(memory bound tree)



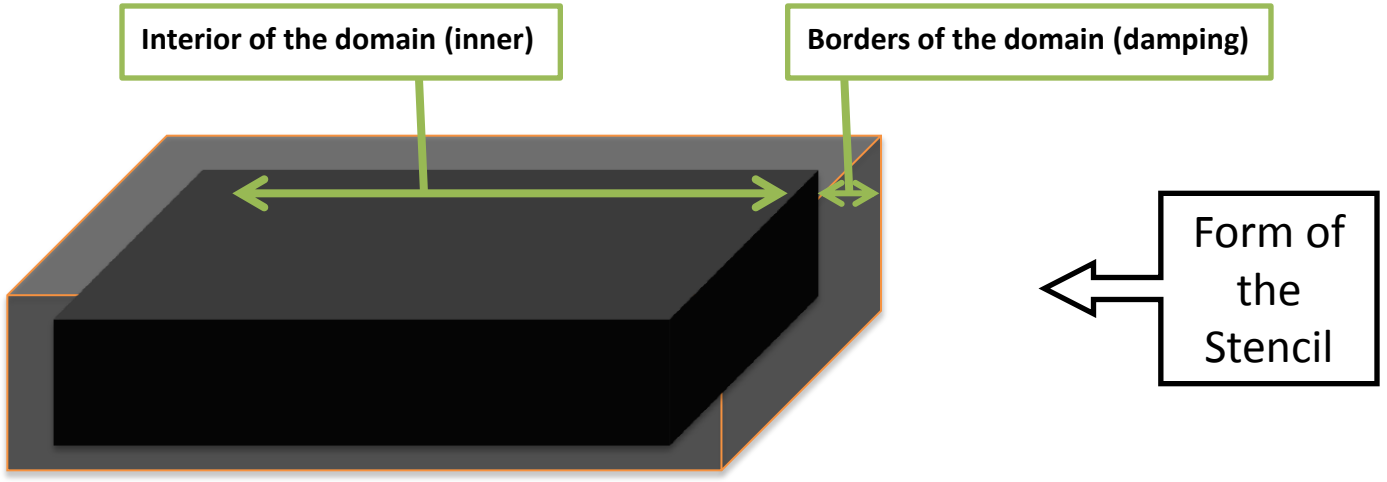
Methodology

(OpenMP tree)



Case study 2: RTM application

- Seismic migration
  - Uses the Reverse Time Migration
- Developed by TOTAL (French oil company)
- Fortran, OMP, MPI, OMP+MPI

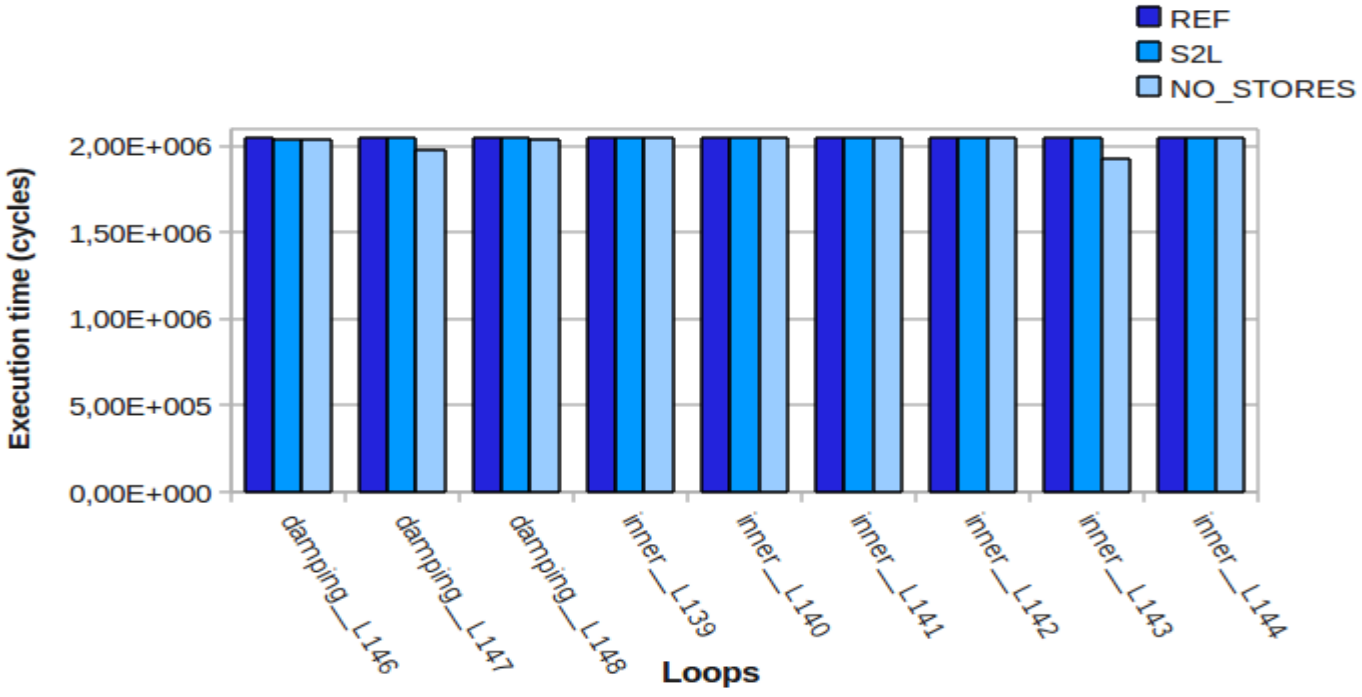


Case study 2: RTM application
(Methodology)

The appliance of the methodology on the code reveals the following:

- Good load balance: equitable work sharing in the stencil
- Good ROI: The chosen blocking strategy provides a reasonable gap between the LS and FP streams. The application is still memory bound
- The last element to check is the possibility of presence of additional coherency traffic => Enabled by the use of the S2L DECAN variant

Case study 2: RTM application
(Cache coherence)



4 cores Sandy-Bridge

Conclusion: Performance are the same => Cache line state change is well managed by the coherency mechanism

MAQAO

- Open source (LGPL 3.0)
  - Currently binary release
  - Source release
- Available for x86-64 and Xeon Phi
- UVSQ member of VI-HPS consortium (www.vi-hps.org)
- Audience
  - User/Tool developer: analysis and optimization tool
  - Performance tool developer: framework services
    - TAU: tau_rewrite (MIL)
    - ScoreP: on-going effort (MIL) interoperability with other tools: Scalasca, Vampir, BSC tools

www.maqao.org

The Stage

Hardware architectures are becoming increasingly complex

- Complex CPU: out of order, vector instructions
- Complex memory systems: multiple levels including NUMA, prefetch mechanisms
- Multicore introduces new specific problems, shared/private caches, contention, coherency
- Each of these hardware mechanisms introduce performance improvement but to work properly, they require specific code properties

Performance pathologies: situations potentially inducing performance loss: hardware poor utilization

- Individual performance pathologies are numerous but finite (they more or less relate to the various architectural features listed above. Non exhaustive of performance pathologies
  - Scalar versus vector
  - Short loop trip count
  - Recurrences
  - Poor spatial locality
  - Poor temporal locality
  - Load imbalance
  - Parallel loop overhead (Fork/join)
  - MPI issues: short messages, early sender/late receiver etc…
- Most of them are well known and well identified

A few major issues with performance pathologies

Some performance pathologies might be quite complex to detect and analyze

- Even for a simple loop nest, pathologies strongly depend upon loop bounds: for example, in general sweeping row wise through an array stored column wise leads to performance problems. However performance penalties can be radically different whether your array is tall (a few columns) versus fat (a large number of columns
- Detection of pathologies can in general be done at the source level, in fact better at the binary level because the compiler can correct some. However some evaluation can be tricky and complex to perform: temporal and spatial locality for example.
- Exact performance impact of a given pathology is hard to evaluate and therefore, the return on investment of correcting is unknown: optimization process advances in the dark.
- Several actors are involved in the performance impact of a performance pathology: algorithm, source code, compiler, OS/runtime and hardware

A major problem with performance pathologies: even for simple loops, several of them might coexist and interact

- Since individual pathologies are finite, their combination remain finite however, from a practical point of view, potential combinatoric explosion of the number of cases to be explored prevents the use of simple exhaustive search techniques
- Interaction between pathologies can be quite complex

Our objectives and approach

OBJECTIVES

- Get a global hierarchical view of performance pathologies/bottleneck
- Get an estimate of performance impact of a given performance pathology taking into account all of the other pathologies present
- Use different tools for pathology detection and pathology analysis
- Perform a hierachical exploration of bottlenecks: the more precise but expensive tools are only used on a specific well chosen cases

General View of our approach

STEP0: offline, microbenchmarck the architecture to get an idea of potential performance impact of different pathologies

STEP1: standard profiling to detect hot routines more precisely the ones contributing up to 80% of total execution time.

STEP2: Value tracing: for all loops analyze loop iteration counts and for array access, array stride (performed via MIL)

STEP3: static analysis via STAN. Assuming operands are in L1, detect and build bottleneck hierarchy

STEP4: split performance problems between OpenMP issues, CPU issues and hierarchical memory issues (use of DECAN)

STEP5: refine data access analysis: perform memory tracing with MTL and group analysis with DECAN

Preliminary step: detecting structural issues

Handling unrolling

- Unrolling can create several binary loops per source loop
- Related binary loops can be clusterized using STAN

Analyzing clusters

- Counting iterations for each binary loop using DECAN
- Using DECAN variant DT1 RAT for the main loop

General observations [may remove this part and transition to the ”Sanity” tree]

- The main loop should process most elements
- Varying iteration count per call may complicate the analysis
- STAN’s cycles prediction should be accurate for DT1 RAT

Introduction
(Contributions)

- Expose a performance assessment methodology based on pathology cost analysis
- Estimate an optimization impact through cost analysis
- Quantitatively determine various pathology impacts on performance

Case study

Dynamic/Static properties after vectorization

- Trip count: from 2 to 2186
- LS (11.75 cycles) vs FP (21.75 cycles): CPU bound => CQA [cycles per source iteration]

Case study

Static properties after vectorization

- DIV/SQRT bound
- Estimated cycles: 21.75 (FP = 21.75)
  - 2x compared to scalar (but 4x elements processed) => 2x speedup
  - First bottleneck: DIV/SQRT
  - Next one: 10.25 cycles (P5 port, probably less if VECTOR ALIGNED)

Case study

Next optimizations

- DIV/SQRT bound (detected):
  - 2x speedup by vectorization
  - More possible by switching to SP (can be combined with Newton Raphson) or reducing number of DIV/SQRT operations
- Inefficient memory accesses:
  - Next bottleneck after DIV/SQRT (and far from it)
  - Optimizations:
    - Using the VECTOR ALIGNED directive
    - Reducing number of strided/indirect accesses

Introduction

(usual performance pathologies)

Pathologies	Issues	Work-around
High number of memory streams	Too many streams for HW prefetcher	See conflict misses
High number of memory streams	Conflict misses	See conflict misses
Lack of loop unrolling	Significant loop overhead, lack of ILP	Try different unrolling factors, unroll and jam for loops nest, try classical affinities (compact, scatter...)

Introduction
(context of performance analysis)

- Lack of ROI
  - Loosing time on optimizing bottlenecks with no significant optimization profit
  - Bottleneck hierarchy: after optimization, how far is next bottleneck
- Parallel analysis
  - Already existing tools (VAMPIR, SCALASCA, VTune, Tau...). But lack of tools for investigating intra-core performance

Methodology

Processor features and parallelism

- Due to some features, lot of performance gain inside each core
  - **Vectorized vs. scalar**: 4x (DP) or 8x (SP) with AVX
  - **ILP**: ADD // MUL // LOAD // STORE (up to 4x)
  - **Memory level**: roughly from 1 (L1) to 20 or more (RAM)
- Possible to emulate parallel workload by assigning synthetic memory load on the remaining cores

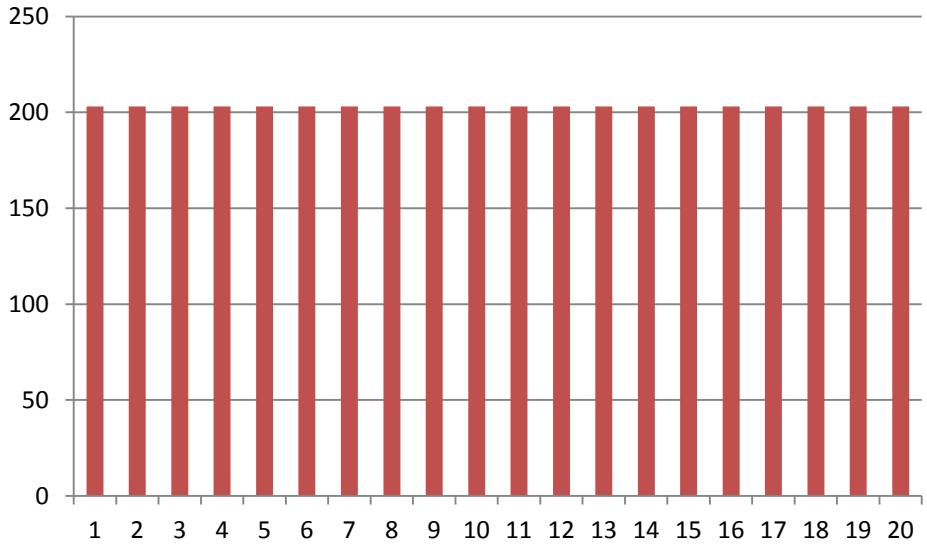
Introduction
(Analysis of current tool set 1/2)

- Some of them cover very well specific but important sub problems
  - MPI issues very well covered for example by TAU, Scalasca, Vampir
  - OpenMP issues are covered but not so well: OpenMP issues very often requires precise tracing and if done at source level, it might be inaccurate (cf Sacalasca)
- Lack of global and accurate view: no indication of performance loss
  - Performance pathologies in general but no hint provided on performance impact (cf VTUNE with performance events): we do not know the pay off if a given pathology is corrected
  - Worse, the lack of global view can lead you to useless optimization: for example, for a loop nest exhibiting a high miss rate combined with div/sqrt operations, it might be useless to fix the miss rate if the dominating bottleneck is FP operations.
  - Source code correlation is not very accurate: for example with VTUNE relying on sampling, some correlation might be exhibited but it is subject to sampling quality and out of order behavior.

Case study

Original code : Dynamic properties (1)

- Trip count: from 1 to 8751 (source iteration count)
- Divide trip count range into 20 equal size interval



All iteration counts are equiprobable (probably triangular access)

Case study

Original code: Static properties

- Estimated cycles: 43 (FP = 44)
- Vector efficiency ratio: 25% (4 DP elements can fit into a 256 bits vector, only 1 is used)
- DIV/SQRT bound + DP elements:
  - ~4/8x speedup on a 128/256 bits DIV/SQRT unit (2x by vectorization + ~2x by reduced latency)
  - Sandy/Ivy Bridge: still 128 bits
  - => First optimization = VECTORIZATION

Case study

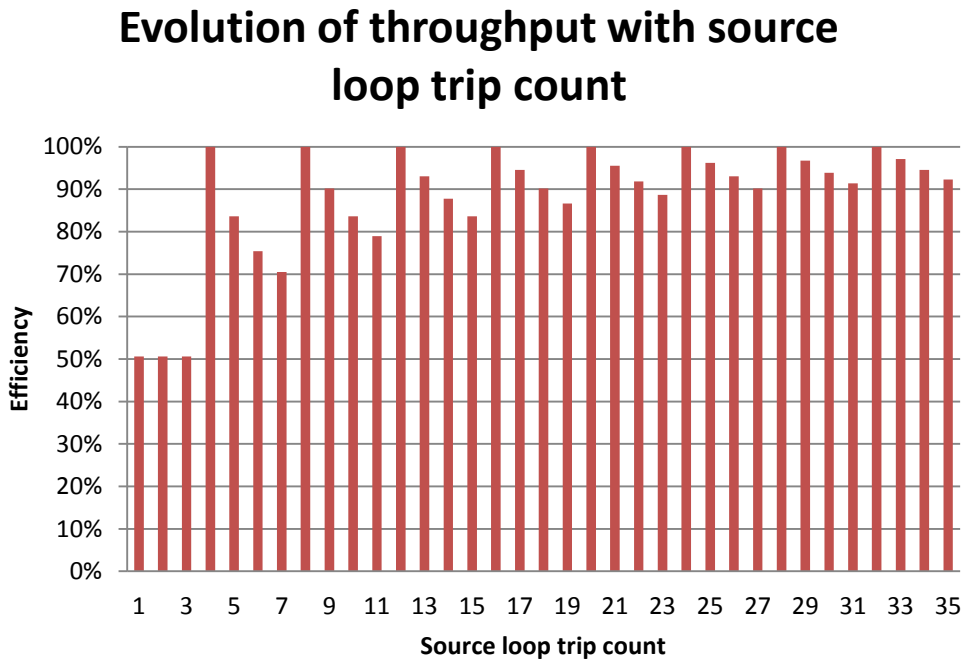
Static properties after vectorization

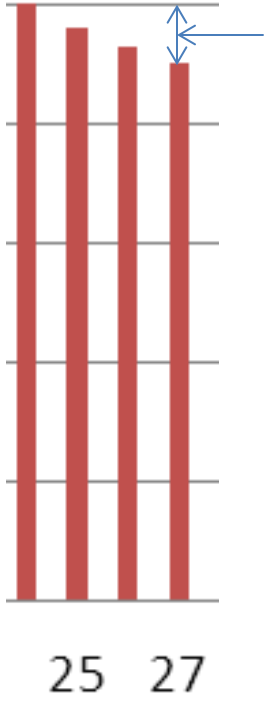
- Vectorization ratio
  - 100% FP arithmetical instructions
  - 65% loads
    - Strided + indirect accesses
    - SCATTER/GATHER not available on Sandy/Ivy Bridge.
- Vector efficiency ratio (vector length usage)
  - 100% FP arithmetical instructions (but 128 bits DIV/SQRT unit)
  - 43% loads (cannot use vector-aligned loads)
  - 25% stores (cannot use vector-aligned stores)

Case study

Static properties after vectorization

- Vectorization overhead: (n/4) x 87 cycles in the main loop vs (n%4) x 43 in the tail loop

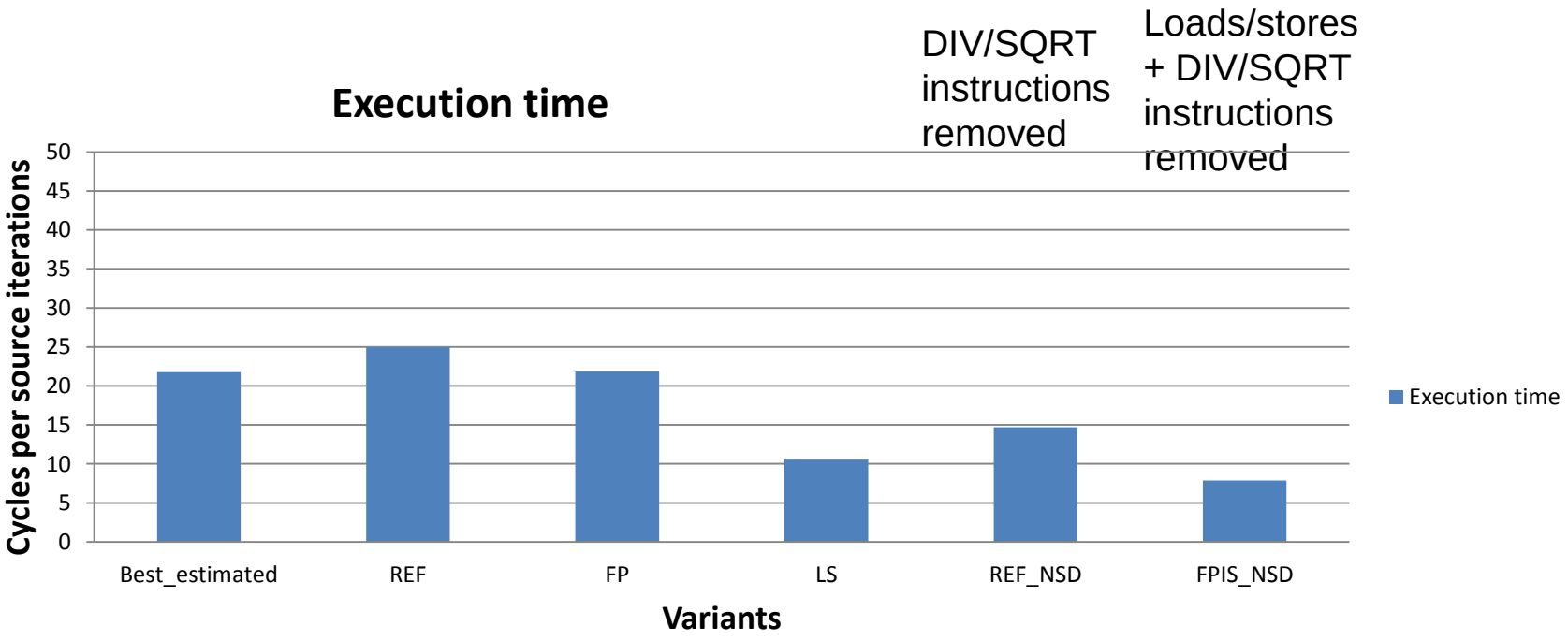




With 27 iterations, 10% of time lost due to 3 iterations in the tail loop

Case study

Dynamic properties after vectorization



Original_NSD: removing DIV/SQRT instructions provides a 2x speedup
=> the bottleneck is the presence of these DIV/SQRT instructions

FP_NSD: removing loads/stores after DIV/SQRT provides a small additional speedup:
next bottleneck

Conclusion: No space for improvement here (algorithm bound)

Tools: *CQA (1/4)*

- CQA = Code Quality Analyzer
- Objectives (provides):
  - Best performance estimation (assuming data in L1)
  - Code quality information (and optimization hints for compiler flags and source transformations)
  - First estimation of bottlenecks hierarchy
- Statically analyzes innermost loops binaries: builds DDG
- Supports Intel 64 micro-architectures from Core 2 to Ivy Bridge (Haswell coming soon)
- Provides metrics and reports at both low and high abstraction levels

Tools: *CQA (2/4, some metrics)*

- Vectorization
  - Vectorization ratio (proportion of vectorized instructions among vectorizable instructions)
  - Vector efficiency ratio [experimental] (average vector length usage of concerned instructions)
  - For not/partially vectorized codes, potential speedup by full vectorization
- Micro-ops and cycles
  - DIV/SQRT and execution ports => bottlenecks
  - Estimated cycles (and derived metrics like GFLOPS)

Tools: *CQA (3/4, low level output)*

Unroll factor: 1 or NA

****
Back-end
****

	P0	P1	P2	P3	P4	P5
FU	*FP ×/÷*	*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%

100% = 256 bits on processors supporting AVX

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

Tools: *CQA (4/4, high level output)*

```
Pathological cases
------------------
Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED. Since your
execution units are vector units, only a fully vectorized loop can use their full power.
By fully vectorizing your loop, you can lower the cost of an iteration from 43.00 to
21.50 cycles (2.00x speedup).
Two propositions:
 - Try another compiler or update/tune your current one:
   * icc: use the vec-report option to understand why your loop was not vectorized. If
"existence of vector dependences", try the IVDEP directive. If, using IVDEP,
"vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
 - Remove inter-iterations dependences from your loop and make it unit-stride.

WARNING: Fix as many pathological cases as you can before reading the following sections.

Bottlenecks
-----------
The divide/square root unit is a bottleneck. Try to reduce the number of division or
square root instructions. If you accept to loose numerical precision, you can speedup
your code by passing the following options to your compiler:
icc: this should be automatically done by default

By removing all these bottlenecks, you can lower the cost of an iteration from 43.00 to
17.00 cycles (2.53x speedup),
```

Case study

Vectorization

- Using SIMD directive

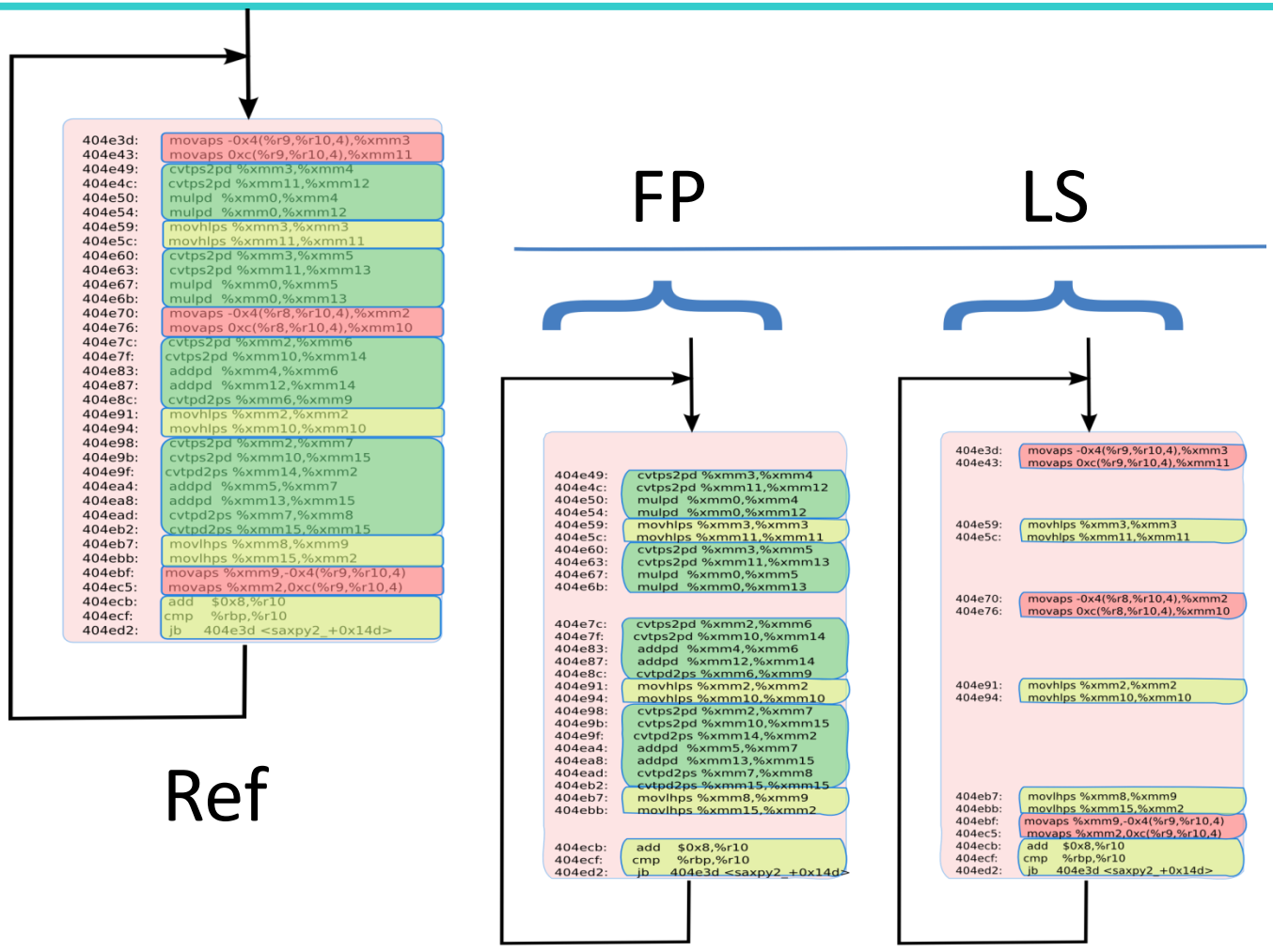
Tools: *Diff. Analysis (1/4), Principles*

- Principle
  - Performance of the original loop is measured
  - Some instructions are removed in the loop body (for example loads and stores)
  - Performance of the transformed loop is measured
- Usage
  - Can perform sampling by transforming only 1 instance and abort execution
  - Can replay original loop execution after modified one
  - The Diff. Analysis speedup is an upper bound for optimization on the removed instructions

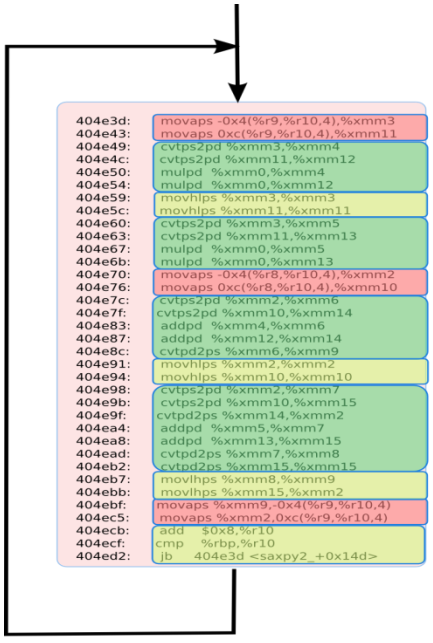
Tools: *Diff. Analysis (2/4), Typical Transformations*

- Key Transformations
  - LS: FP operations are suppressed (or replaced with NOPS)
  - FP: Load Store operations are suppressed (or replaced with NOPS)
  - DL1: All of the Load/Store operations target operands in L1
  - NoDiv (No Sqrt): DIV (SQRT) operations are suppressed or replaced with NOPS
  - NoRecur: all of the inter iterations dependencies are suppressed
  - S2L: Stores are replaced with Loads tageting the same address.
- REMARKS
  - Transformations can be combined
  - All of the loop control instructions are laways preserved.

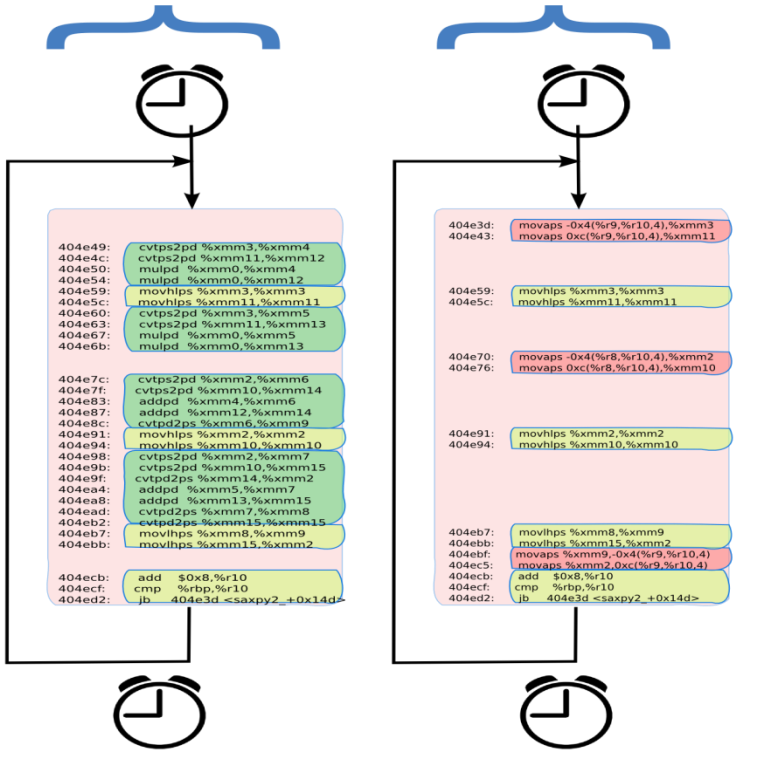
Tools: *Diff. Analysis (3/4, FP, LS transformations)*



Tools: *Diff. Analysis (4/4, FP, LS transformations)*



Monitor
- Execution times
- Loop Iteration numbers
- hardware counter values



Yales2 codelet clustering

Cluster 1
14 codelets – Coverage: 2,13%
AXPY

Cluster 2
10 codelets – Coverage: 3,8%
AXPY + Accum

Cluster 4
2 codelets – Coverage: 1,19%
Behaviour
ics_advance_velocity_tfv4a_4th

Cluster 12
18 codelets – Coverage: 7,97%
IPC > 3,5

Array copy

Cluster 5
33 codelets – Coverage: 8,33%
low byte stored/cycle
High P1

Cluster 18
9 codelets – Coverage: 1,40%
High byte stored/cycle
Low P1

Cluster 9
12 codelets – Coverage: 1,03%
Best array copy efficiency