Exascale computing research

# *DECAN: Differential Analysis for fine level performance evaluation.*

Current contributors: E. Oseret (ECR), M. Tribalat (ECR), C. Valensi (ECR), W. Jalby (ECR/UVSQ)

Former DECAN contributors: Z. Bendifallah (ATOS), J.-T. Acquaviva (DDN), T. Moseley (Google), S. Koliai (Celoxica), J. Noudohouenou (INTEL),



# OUTLINE

- Application developer point of view
- DECAN: principles
- A motivating example
- DECAN: general organization
- Conclusions

# Application developer problems (1)

- First, a **larger** number of **ever more complex** hardware mechanisms (more FU, more caches, more vectors etc …) are present in modern architectures
  - Each of these mechanisms might be a potential performance bottleneck!!
- To get top performance all of these mechanisms have to be fully exploited
- **Code optimization has become a very complex task:**
  - Checking all of these potential sources of performance losses (poor exploitation of a given resource: performance pathologies)
  - Checking potential dependences between performance issues
  - Resolving chicken and egg problem: program run out of physical register files due to long latency operations such as divide
  - Building pathology hierarchy: what are the most important issues which have to be worked first….

# Application developer problems (2)

**Classical technique of working first on the loop with the highest coverage (contribution) is not a valid strategy**:

- Importance of ROI (Return On Investment)
  - Routine A consumes 40% of execution time and performance gains are estimated on routine A at 10%: overall gain 4%
  - Routine B consumes 20% of execution time and performance gains are estimated on routine B at 50%: overall gain 10%

**WORK FIRST ON B (NOT A) BUT REQUIRES EVALUATING ACCURATELY PERFORMANCE GAINS:**

- **Knowing number of cache misses is not enough**
- **Knowing cache miss latency is not enough either…**
- **We need to know performance impact of a cache miss: much more subtle notion and how to measure it…. In fact, you would like to be able to "suppress" cache misses and measure performance..**
- **Evaluation of "What if" scenarios. Most of the current analysis techniques measure what happens, never what could have happened if …**

# Application developer problems (3)

The main knobs that an application developer can use for tuning are:

- Modify source code
- Write in assembly ☺
- Insert directives
- Use compiler flags

**To use most of these knobs, very good correlation has to be established between performance problems and source code, ultimately at the source line level.**

**In addition to the previous info on cache misses, we also need to know what array(s) access are generating these misses….**

**How to get all of that info ??**
**Main goal of DECAN and differential analysis**

# DECAN Principles (1)

- Be a physicist:
  - Consider the machine as a black box
  - Send signals in: code fragments
  - Observe/measure signals out: time and maybe other metrics
- Signals in/Signals out
  - Slightly modify incoming signals and observe differences/variations in signals out
  - Tight control on incoming signal
- In coming signal: code
  - Modify source code: easy but dangerous: the compiler is in the way
  - Modify assembly/binary: much finer control but more complex and care about correlation with source code

# DECAN Principles (2)

- GOAL 1: detect the offending/delinquent operations
- GOAL 2: get an idea of potential performance gain

# DECAN Principles (3)

DECAN's concept is simple:

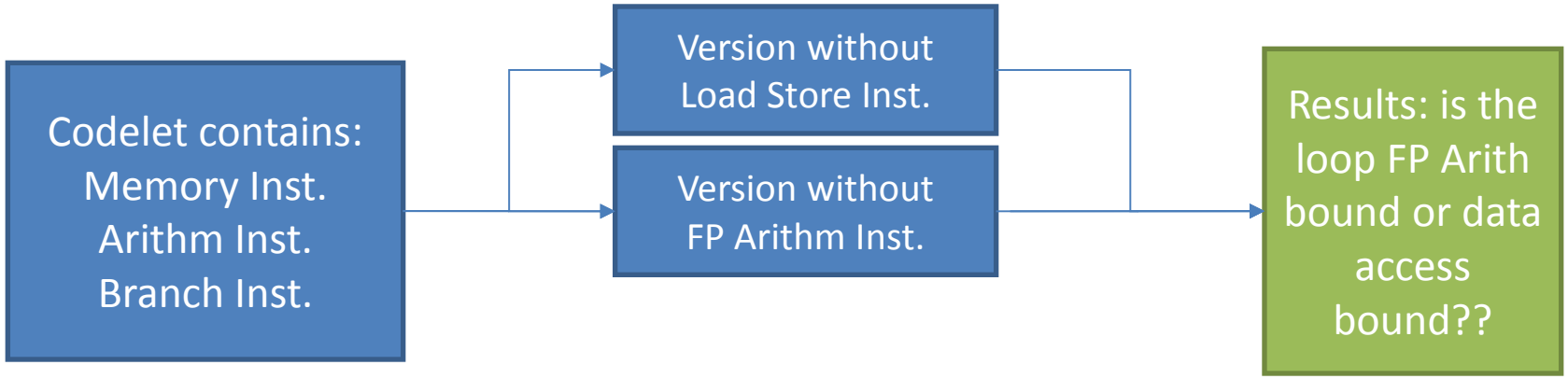
- Measure the original binary
- Patch the target instruction(s) in the original binary
- New binaries are generated for each patch
- Measure new binaries
- Compare measurements and evaluate instruction cost differentially

CLEAR NEED: manipulate/transform/patch binaries

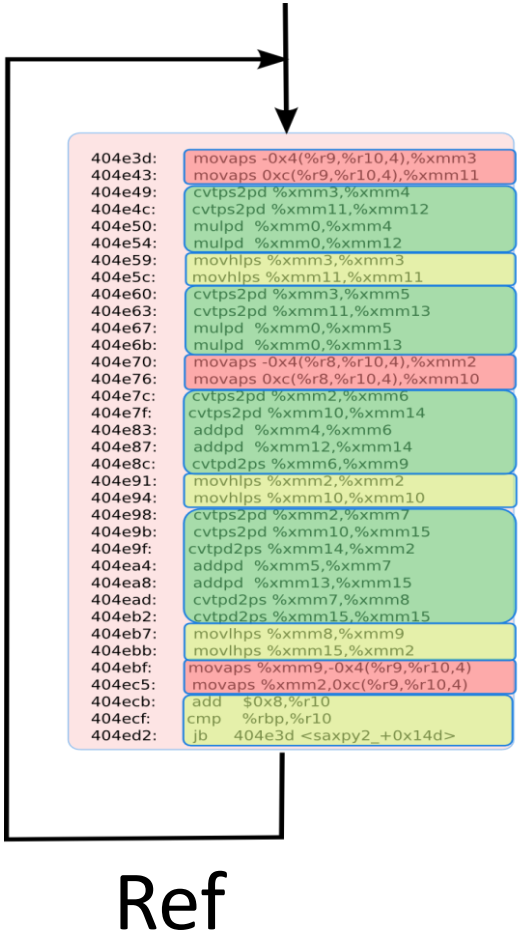
# DECAN: SIMPLE VARIANTS (1)

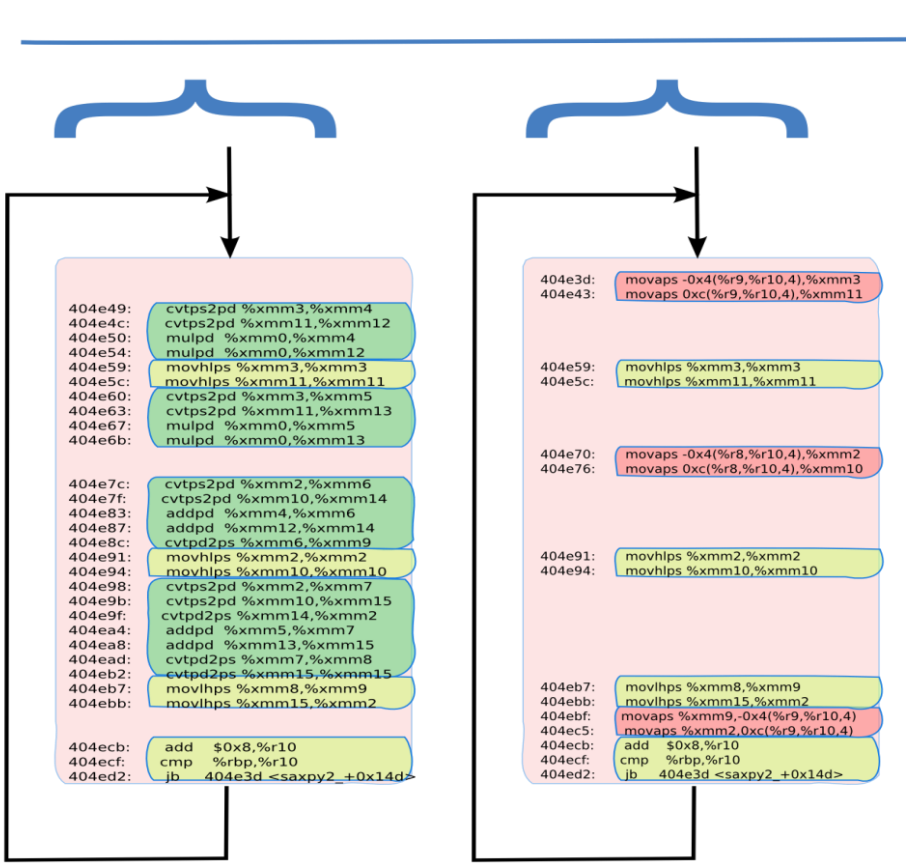
DECAN generates binary variants according to predefined templates/rules

- FP: all of the SSE/AVX instructions containing Load/Stores are removed
- LS: all of the SSE/AVX instructions containing FP arithmetic are removed

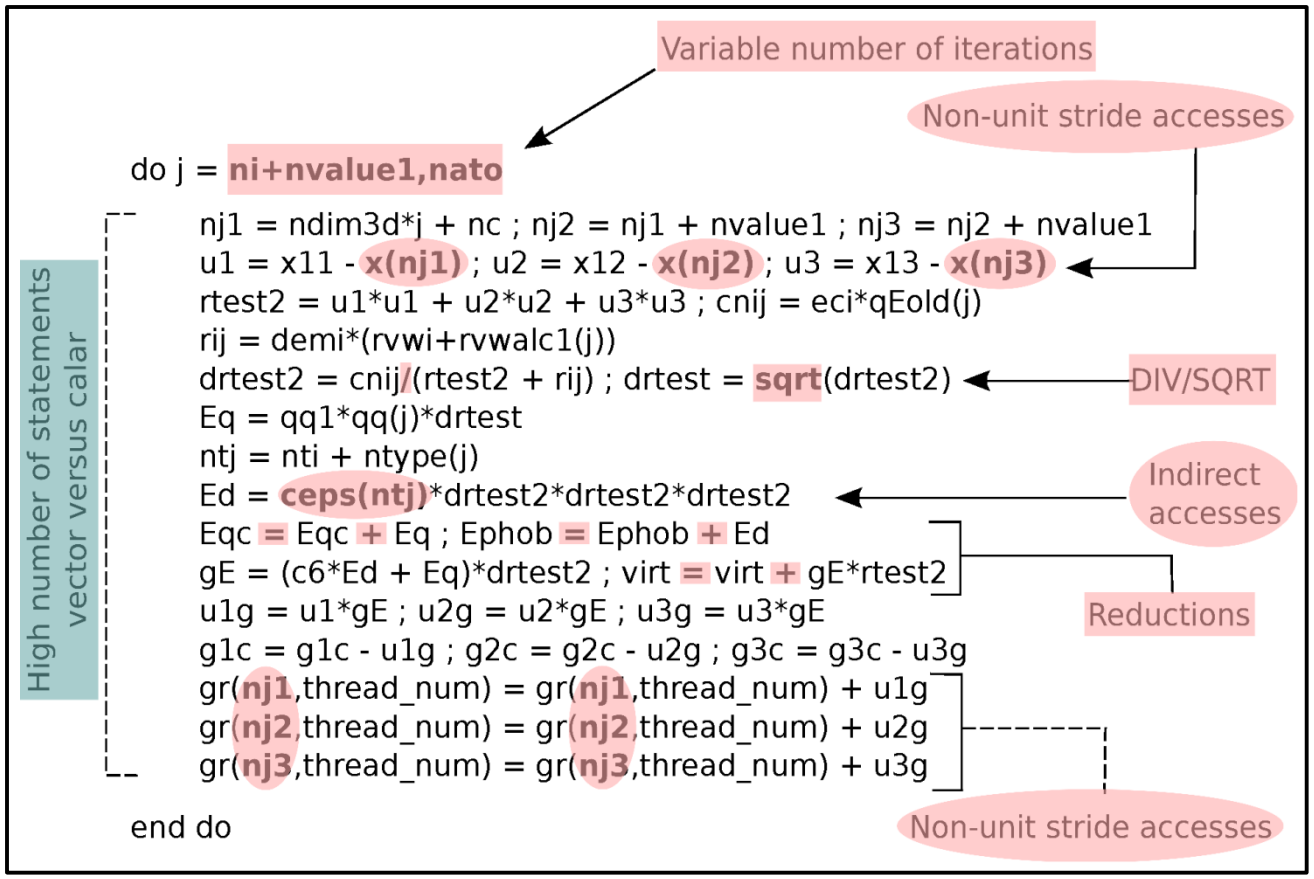


# DECAN: SIMPLE VARIANTS (2)



# Motivation: *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 ?

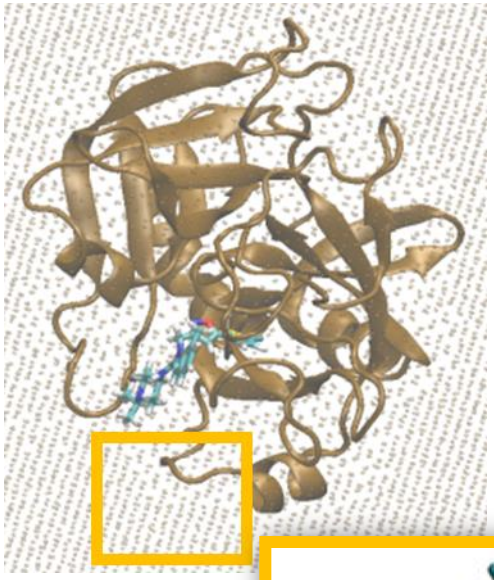


11

# Motivation: *POLARIS(MD) Loop*

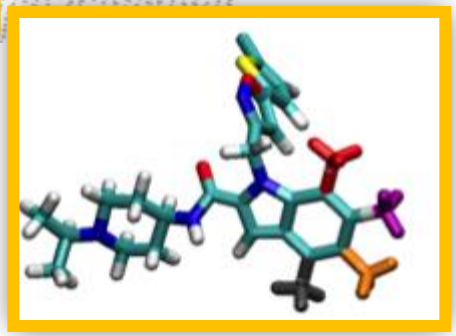
- Molecular Dynamics
  - Based on the Newton equation: $m\vec{a} = -\vec{\nabla}U_{pot}$
  - Multiscale
- Developed at CEA (French energy agency) by Michel Masella
- 60K LOC, Fortran 90
- OMP, MPI, OMP+MPI

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



$(7.46\ nm)^3$

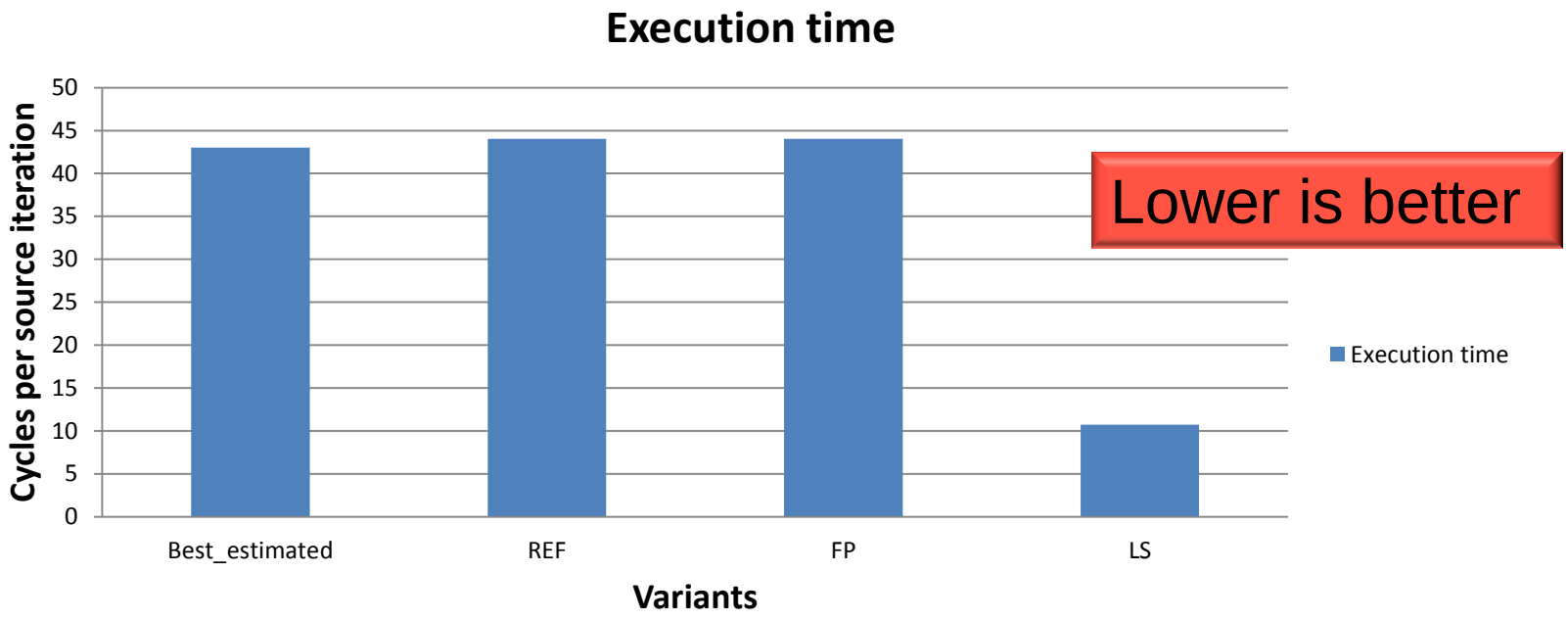
Anti-Coagulant





12

# Original Code: Dynamic properties

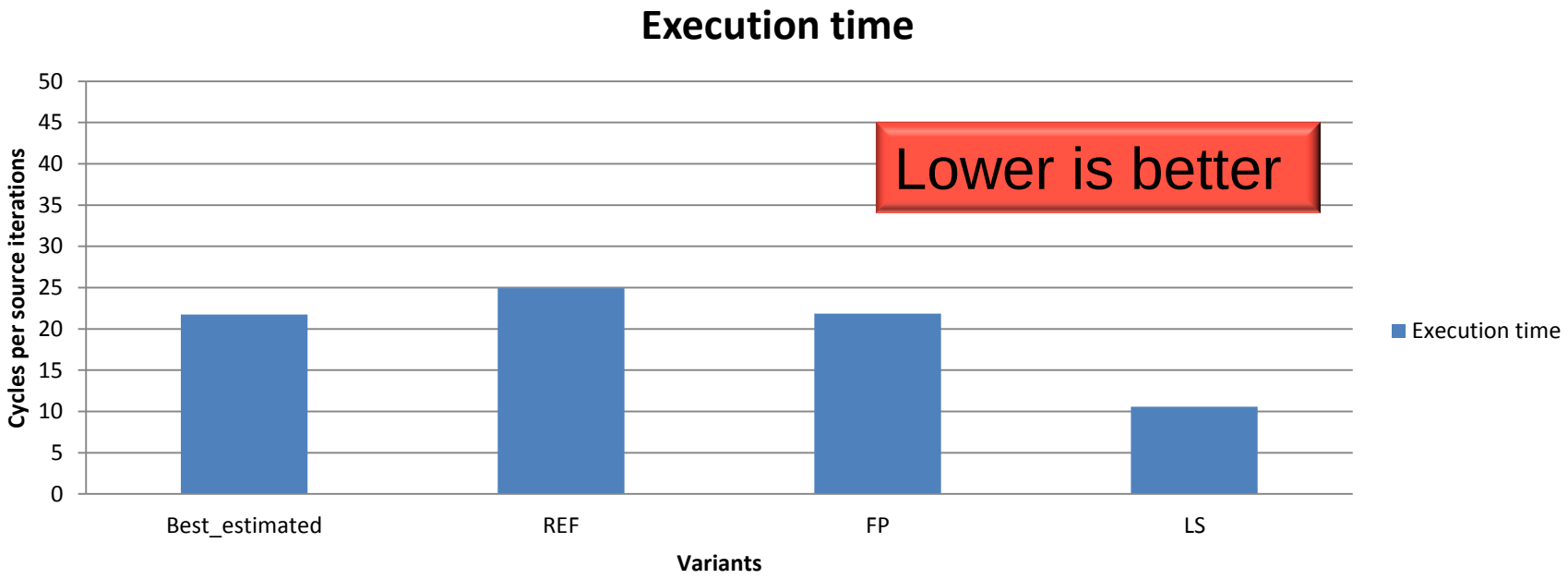



**TARGET HARDWARE: SNB**

- **Best Estimated:** CQA (static Code Quality Analyzer) 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: Work on FP**
- **CQA indicates DIV/SQRT major contributor. Let us try to vectorize DIV/SQRT**

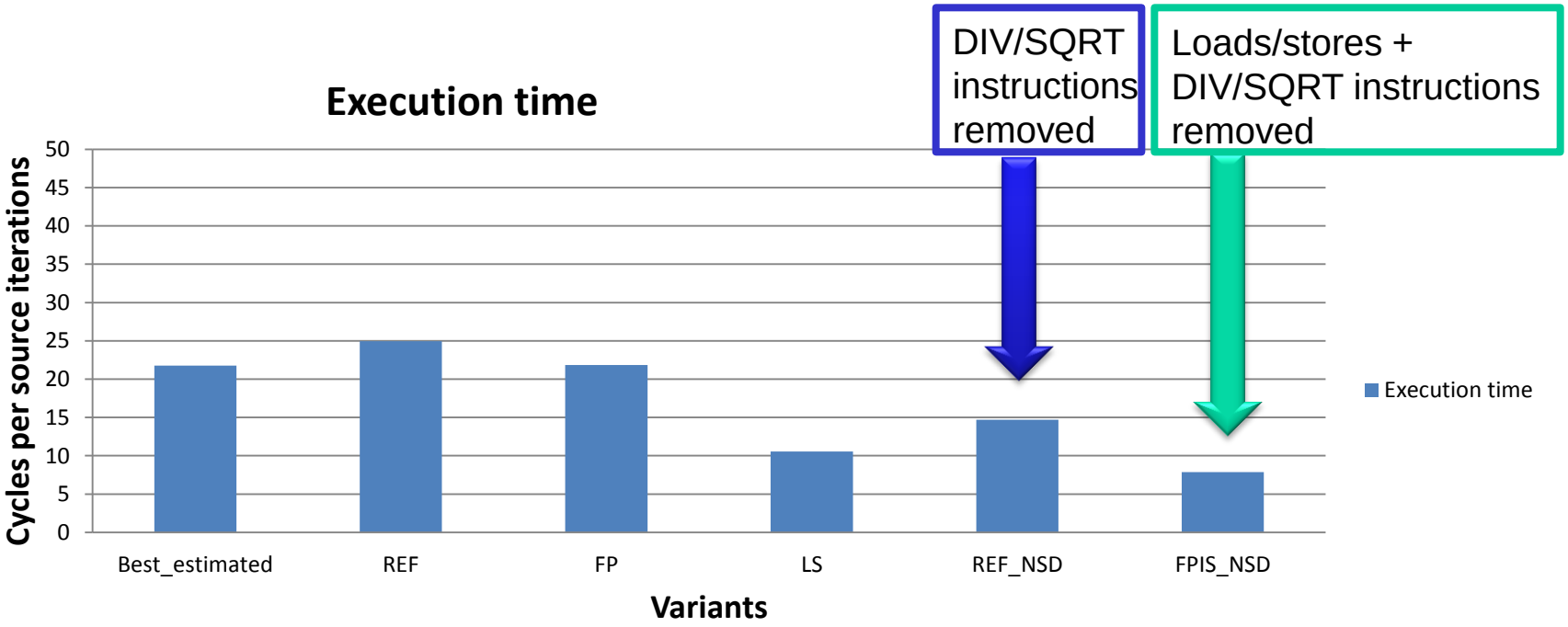
# Vectorized Code: properties update

## Forced vectorization using SIMD directive.




- FP / LS = 4,1 ⟹ 2,07
- REF: 45 ⟹ 25
- FP: 44 ⟹ 22
- LS: 10 ⟹ 10

# Case study: *one step further*




**REF_NSD** : removing DIV/SQRT instructions provides a 1.5 x 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)

# Loop Variant creation

Identify target instruction subsets → Construct transformations requests → Inject monitoring probes

## Examples of instruction subsets

- Load & Store
- Load
- Store
- Adress Comput
- Control Flow
- FP arithmetic
- Division
- Reduction
- ……

## Examples of transformations

- Deletion
- Replacement
- Modification

## Observed events

- Time
- PMU events

Transformations done independently on every instruction in the subset.
Control flow instructions are blacklisted and never affected by transformations

# How to use DECAN while preserving semantics

DECAN variant execution will provide incorrect results. DECAN variants are inserted in the binary using the following process.

- Context Save
- Start RDTSC (or other probe)
- DECAN Variant (FP, LS, etc…) execution
- Stop RDTSC (or other probe)
- Restore context
- Original loop execution
- Resume regular program execution

# DECAN: Coarse Performance Analysis

- Comparing LS and FP measurements allows to detect whether
  - The loop is data access bound then work has to be done on data access
  - The loop is FP bound then work on vectorization, removing long latency instructions etc ...
  - DECAN has provided us a clear performance estimate gain.
- We need to go further and start working on individual instructions or better groups of instructions:
  - Suppress all loads
  - Suppress all stores

REMARK: suppressing a single instruction can be hard to interpret.

# DECAN: Parallelism

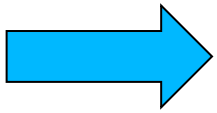
DECAN can be used for unicore code but also for parallel constructs:

- Data parallel, DOALL OpenMP loops can be DECANNed: all of the threads will execute the same modified binary load/store instructions corresponding to G are suppressed
- Same technique can be used for MPI code although care has to be taken on the core use of the memory.
- Issue: analyzing results with a large number of threads.

# Impact of Operand Location: Emulating Perfect L1

## Original ASM

```
Loop:
vmovupd (%rdx,%r15,8), %ymm4
vmovupd (%rdx,%r15,8), %ymm5
vaddpd %ymm4, %ymm5, %ymm6
vmovupd %ymm6, (%rax,%r15,8)
add $4, %r15
cmp %r15, %r12
jb Loop
```



## Modified ASM: DL1

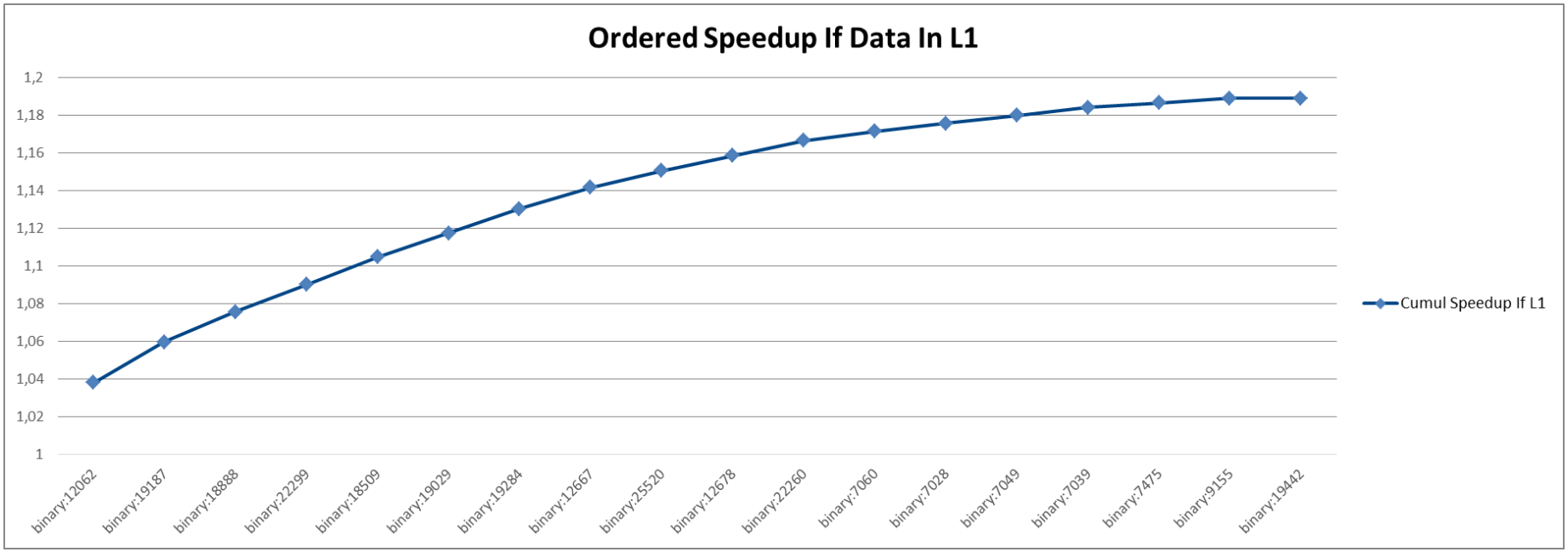
```
Loop:
vmovupd a(%rip), %ymm4
vmovupd a(%rip), %ymm5
vaddpd %ymm4, %ymm5, %ymm6
vmovupd %ymm6, a(%rip)
add $4, %r15
cmp %r15, %r12
jb Loop
```

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

# ONE VIEW R.O.I. DL1

- Results showing the potential speedup if all data was in L1 cache for the YALES 2 application (*3D Cylinder* model)
- Loops ordered by coverage.



# How to use DECAN in a systematic manner

- ***Use various tools (sampling, tracing, static analysis)***
  - CQA for analyzing code quality
  - Sampling to estimate loop coverage
  - Value profiling (tracing) to get loop iteration count
- ***Integrate DECAN variants in a decision tree similar to the Top Down Approach proposed by Yasin et al***
- ***PAMDA: Performance Analysis Methodology using Differential Analysis***

# DECAN limitations

- DECAN is complex: side effects have to be analyzed with care in particular when using new variants
- Dependent upon code generated/compiler: loops with multiple entry points ??
- DECAN is a microscope: applicable to loops only
  - Needs to be coupled with good profiling
- Measurement accuracy
  - Let us think of a loop with 100 groups (each of them accessing a different array): suppressing one group might be equivalent to suppress 1% work, hard to detect.
  - Some experiments in the DECAN series can crash: for example NOP the access to indirection vectors

# DECAN Conclusion

- DECAN is a powerful tool for
  - Detecting performance bottlenecks
  - Evaluating performance potential gains
  - Providing correlation between source code and performance issues
- DECAN only needs a precise timer even for analysing memory behavior.
- DECAN integrated with ONE VIEW tool set used by CEA DAM, CEA Life Science (POLARIS MD), CERFACS (AVBP), Dassault Aerospace, INTEL ECR, …
- DECAN is Open Source (LGPL 3.0)

# BACKUP SLIDES

# DECAN: Dealing with Branches

## Dealing with If within loop bodies

- Typical case: if (A(I)) > 0) THEN  (BBBBB) ELSE (CCCC)
- First analysis: preserve loop control and apply transformations on (BBBBBB) and (CCCCC)
- Second analysis: Suppressing access to A(I) is equivalent to NOPping the branch. Can be used to analyze cost of mispredicts
- DECAN provides info but care has to be taken

# LS/FP Variants

## LS variant

Arithmetic operations are deleted

```
MOVAPS   %XMM2,(%R9,%R8,8)
MOVAPS   0x10(%RDI,%R8,8),%XMM3
MULPD    %XMM1,%XMM3
ADDPD    0x10(%R9,%R8,8),%XMM3
```

```
MOVAPS   %XMM2,(%R9,%R8,8)
MOVAPS   0x10(%RDI,%R8,8),%XMM3

MOVAPS   0x10(%R9,%R8,8),%XMM3
```

## FP variant

- Memory operations are deleted

```
MOVAPS   %XMM2,(%R9,%R8,8)
MOVAPS   0x10(%RDI,%R8,8),%XMM3
MULPD    %XMM1,%XMM3
ADDPD    0x10(%R9,%R8,8),%XMM3
```

```
MULPD    %XMM1,%XMM3
ADDPD    %XMM3,%XMM3
```

## Effect

- CPU and memory sub-system behavior highlighted independently

# Tools: *CQA*

***CQA = Code Quality Analyzer***
***Objectives (provides):***

- Statically analyzes innermost loops binaries: builds DDG
- Best performance estimation (assuming data in L1 and using microbenchmarks for FU latencies/bandwidths)
- Code quality information (and optimization hints for compiler flags and source transformations)
- First estimation of bottlenecks hierarchy
- Provides metrics and reports at both low and high abstraction levels
- Supports Intel 64 micro-architectures from Core 2 to Coffee Lake


28

# Side Effects to Monitor (1)

| Side effect | Workarounds |
|---|---|
| Code layout change | Replace deleted instructions with NOPs |
| Data dependency | Kill extra dependencies introduced |
| Variable latency instructions | Control latency by loading the operands |
| Floating point exceptions | Deactivate software exception handling |
| Different floating behavior | Load special values from stack |

# Side Effects to monitor (2)

Suppressing load store instructions can introduce extra (unwanted) dependencies:

*ADDPD (%rsi), xmm1*
*MOVAPS xmm1, 16(%rsi)*
*MOVAPS (%eax), xmm1*
*ADDPD xmm2, xmm1*

Is transformed into (adding PXOR allows to break dependencies):

*ADDPD (%rsi), xmm1*
*XORPS xmm1, xmm1*
*ADDPD xmm2, xmm1*

# Coherency Impact Analysis

## S2L variant

Transform every store operation into a load operation with same target adress

```
MOVAPS  %XMM2,(%RDI,%R8,8)
```

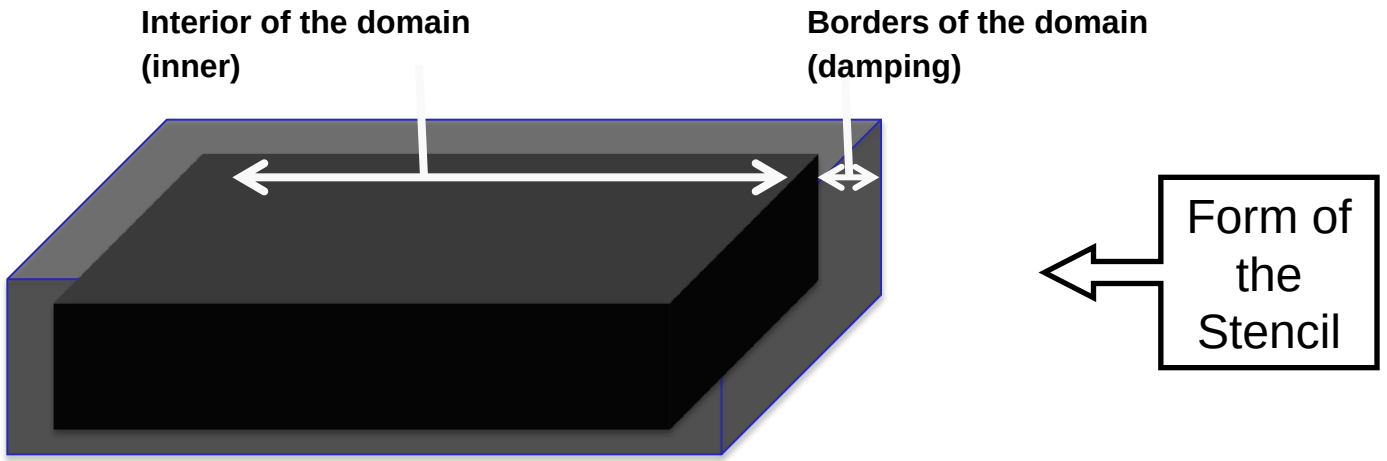
```
MOVAPS  (%RDI,%R8,8),%XMM2
```

## Effect

Disables all the cache effects caused by stores (coherency issues)

# RTM code (1)

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



# RTM code : OpenMP study

Preliminary performance studies:

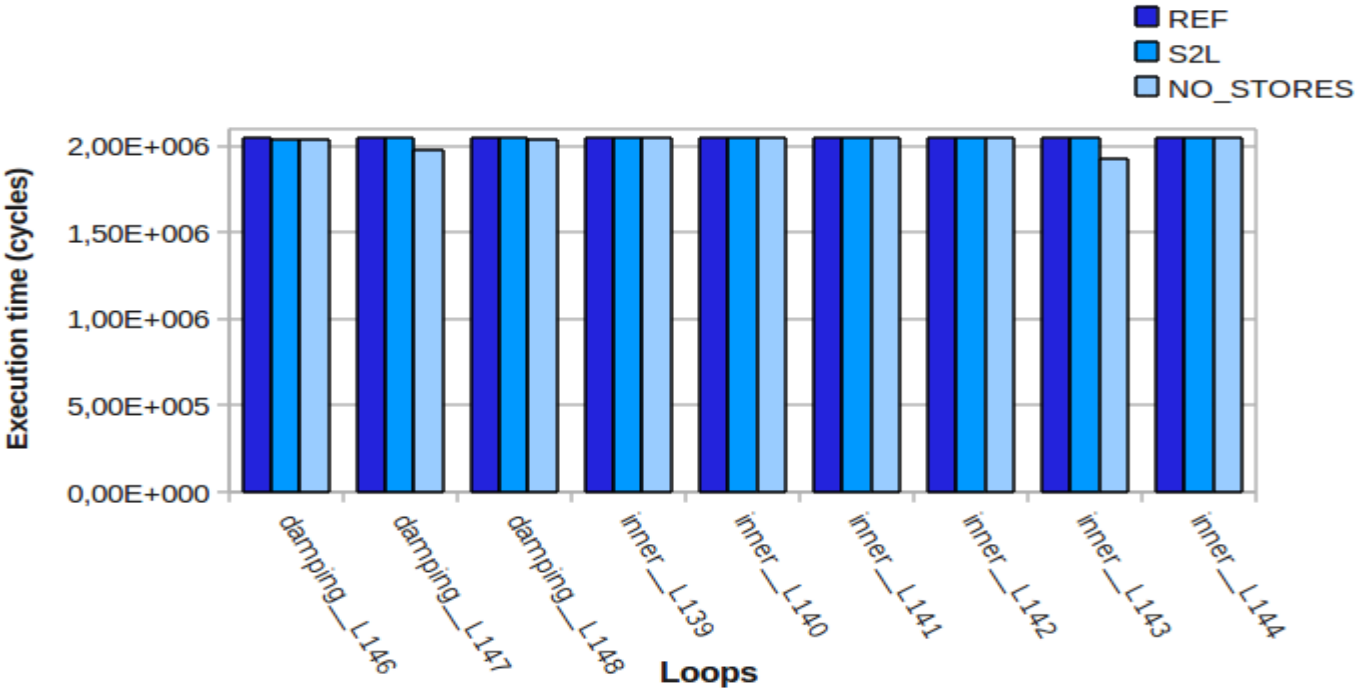
- **Good load balance:** equitable work sharing in the stencil

- **Good locality**: The chosen blocking strategy provides a reasonable gap between the LS and FP streams. The application is still memory bound

Due to OpenMP parallelization strategy (subdomain decomposition), many elements are written by cores then read by other cores. Potential data coherence traffic issue.

**Use S2L DECAN variant!!**

# RTM application *(Cache coherence )*

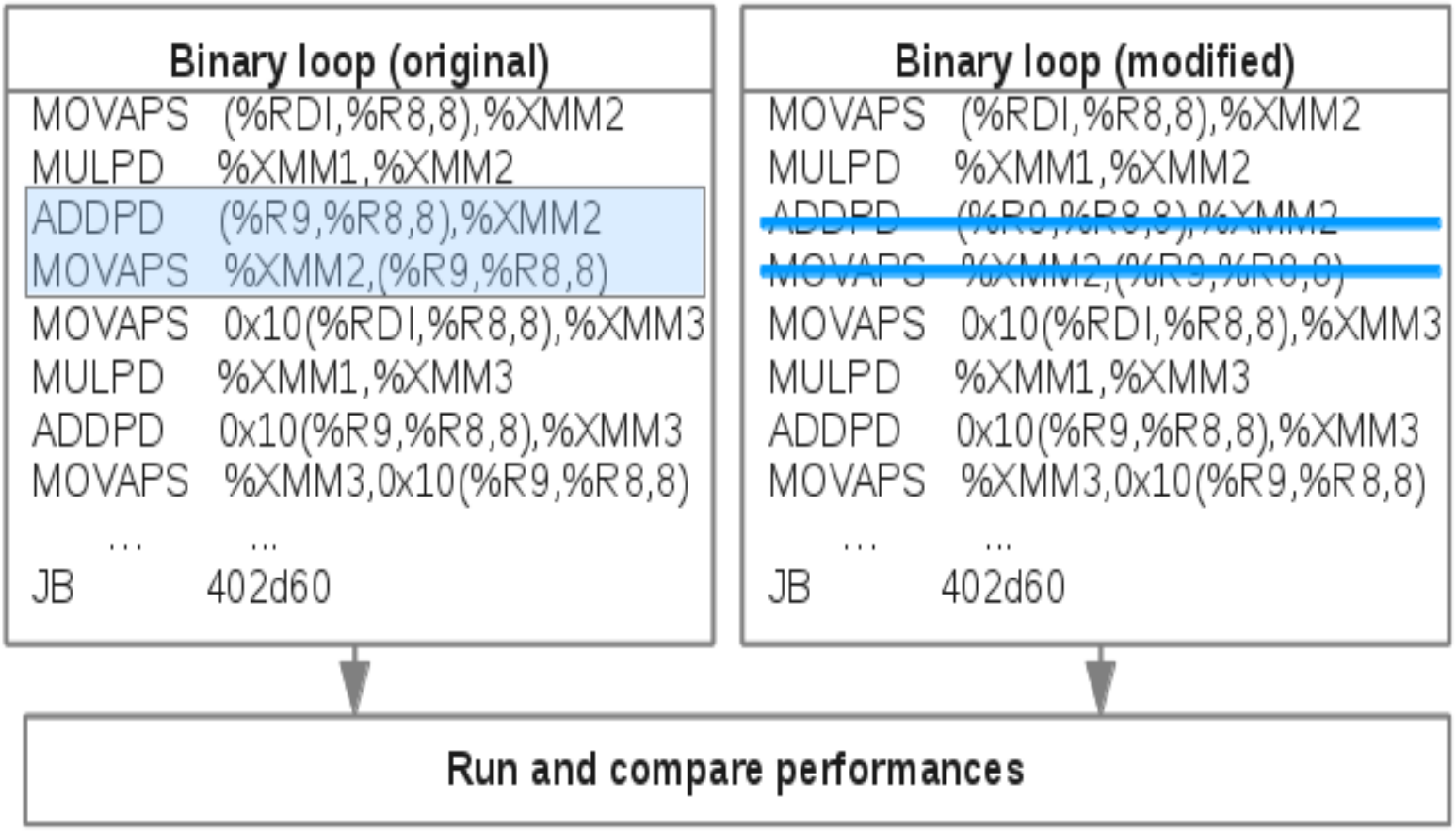



4 cores Sandy-Bridge

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

# Decremental Analysis: a first example

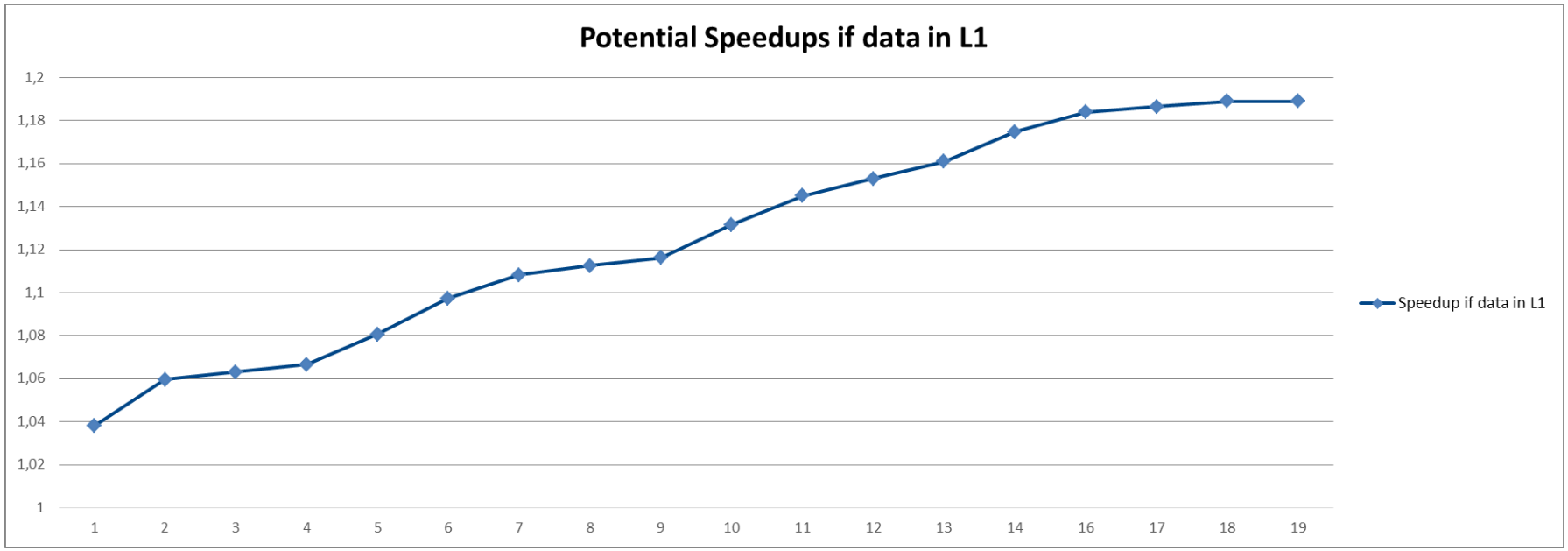
A measurement technique based on binary program modification



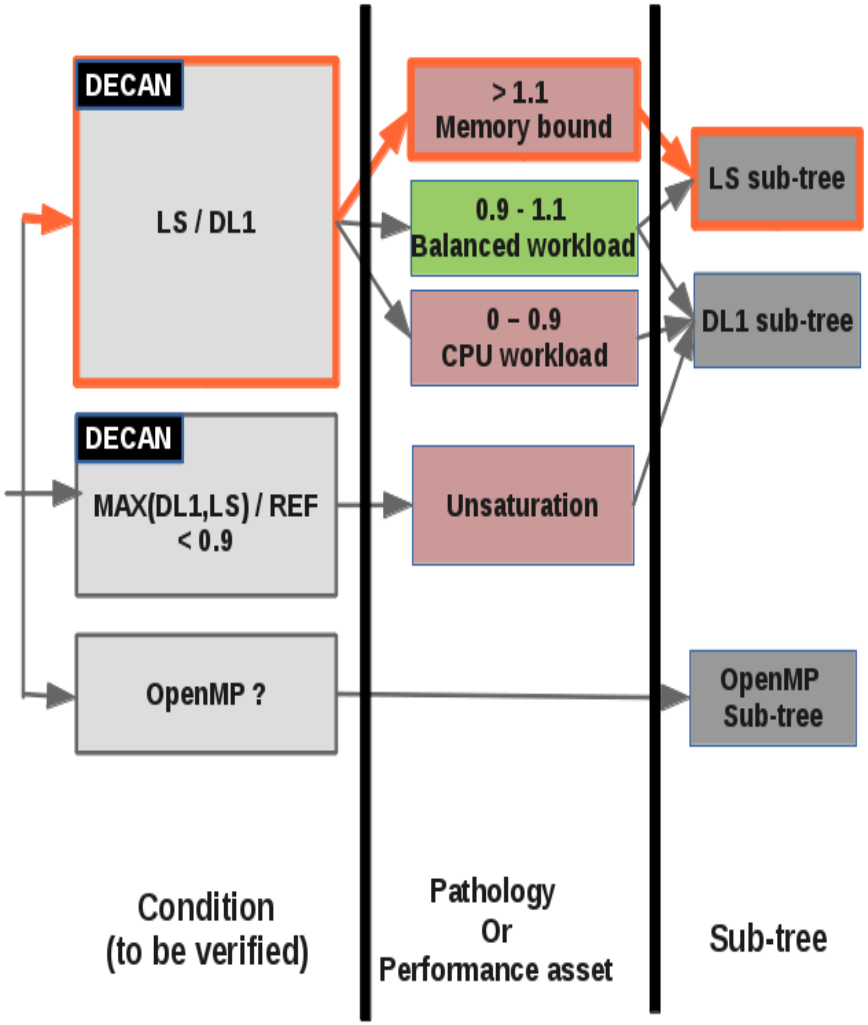

Modified binary is wrong: produces erroneous results

# ONE VIEW R.O.I. DL1

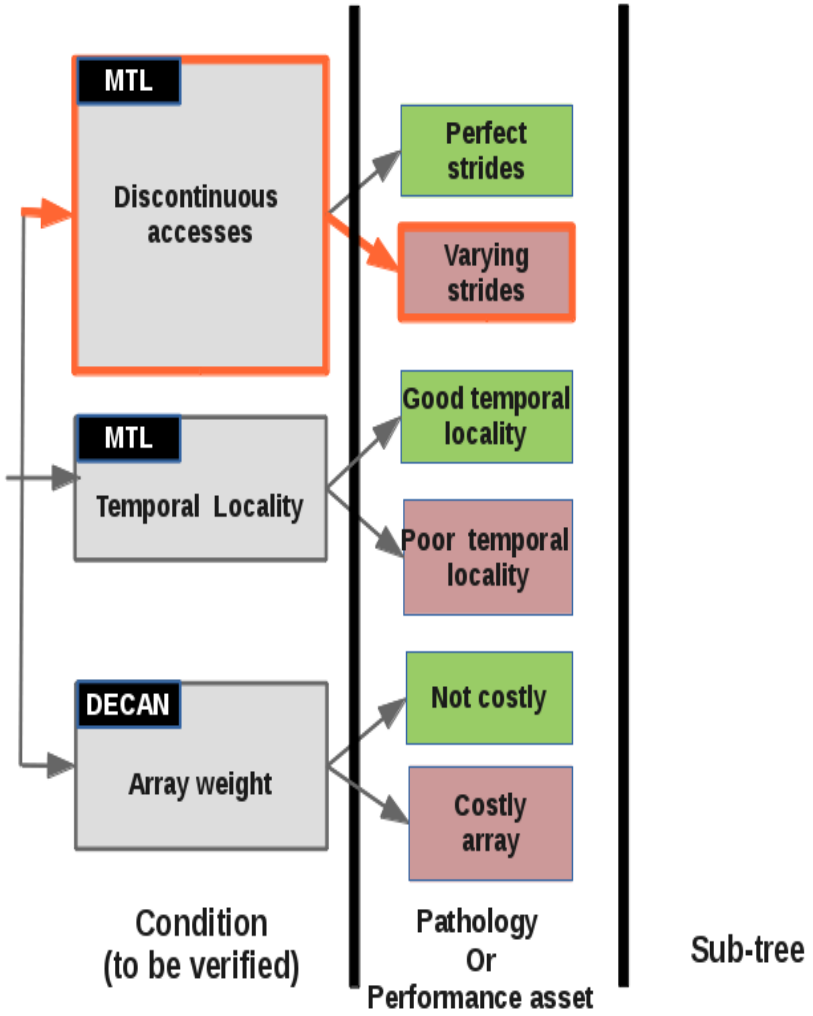
- Results showing the potential speedup if all data was in L1 cache for the YALES 2 application (*3D Cylinder* model)
- Loops ordered by coverage.



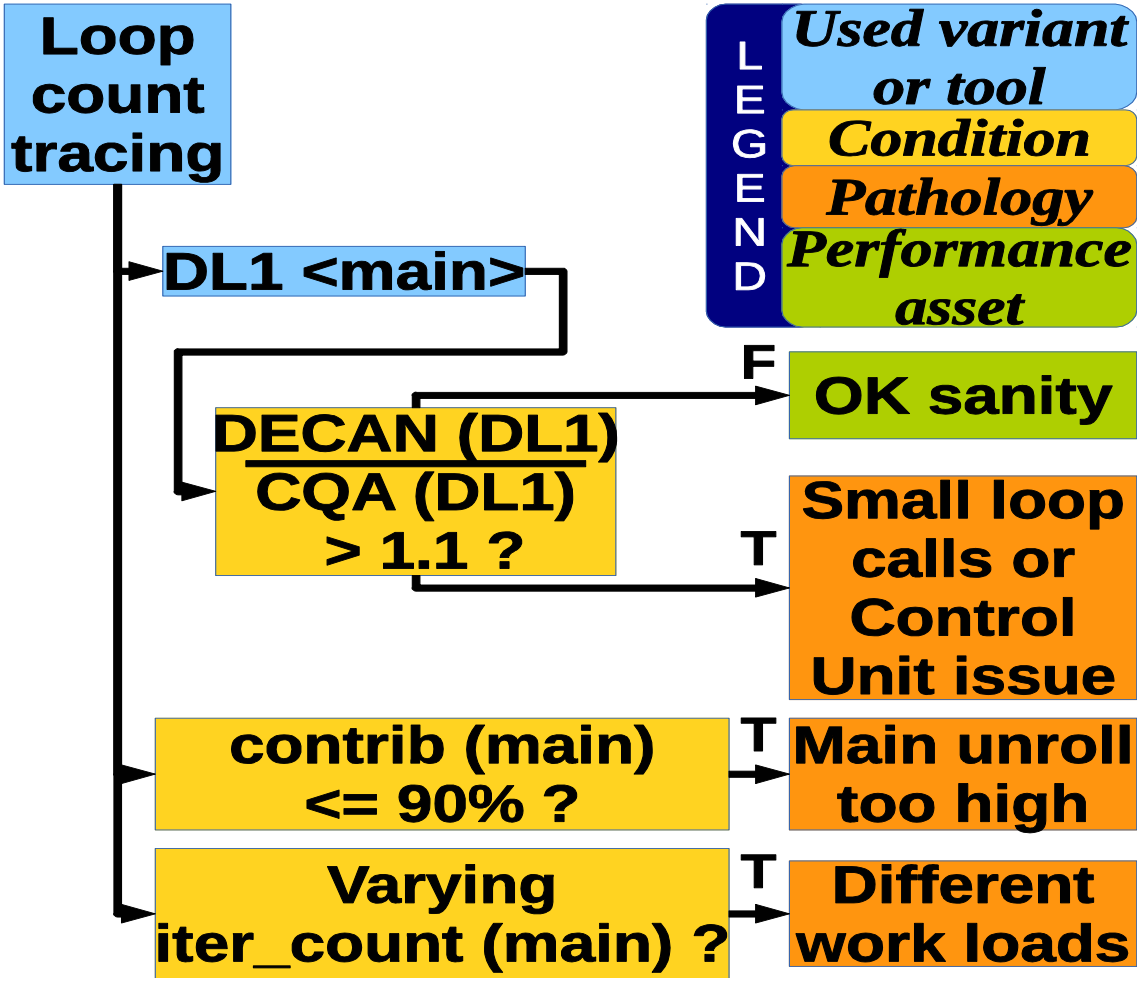
# PAMDA: Global Decision Tree



# PAMDA: LS Sub Tree



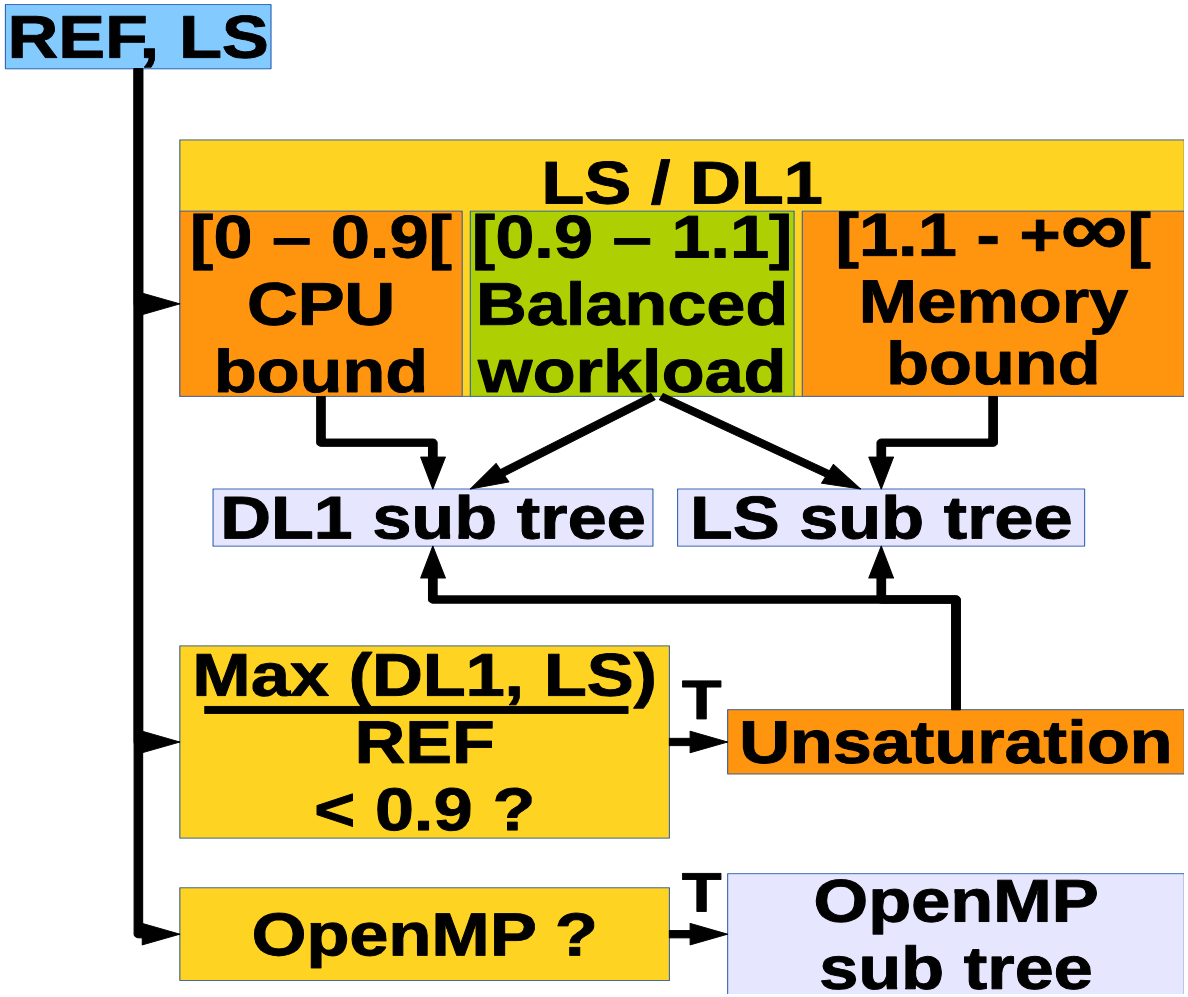
# Methodology *(sanity tree)*



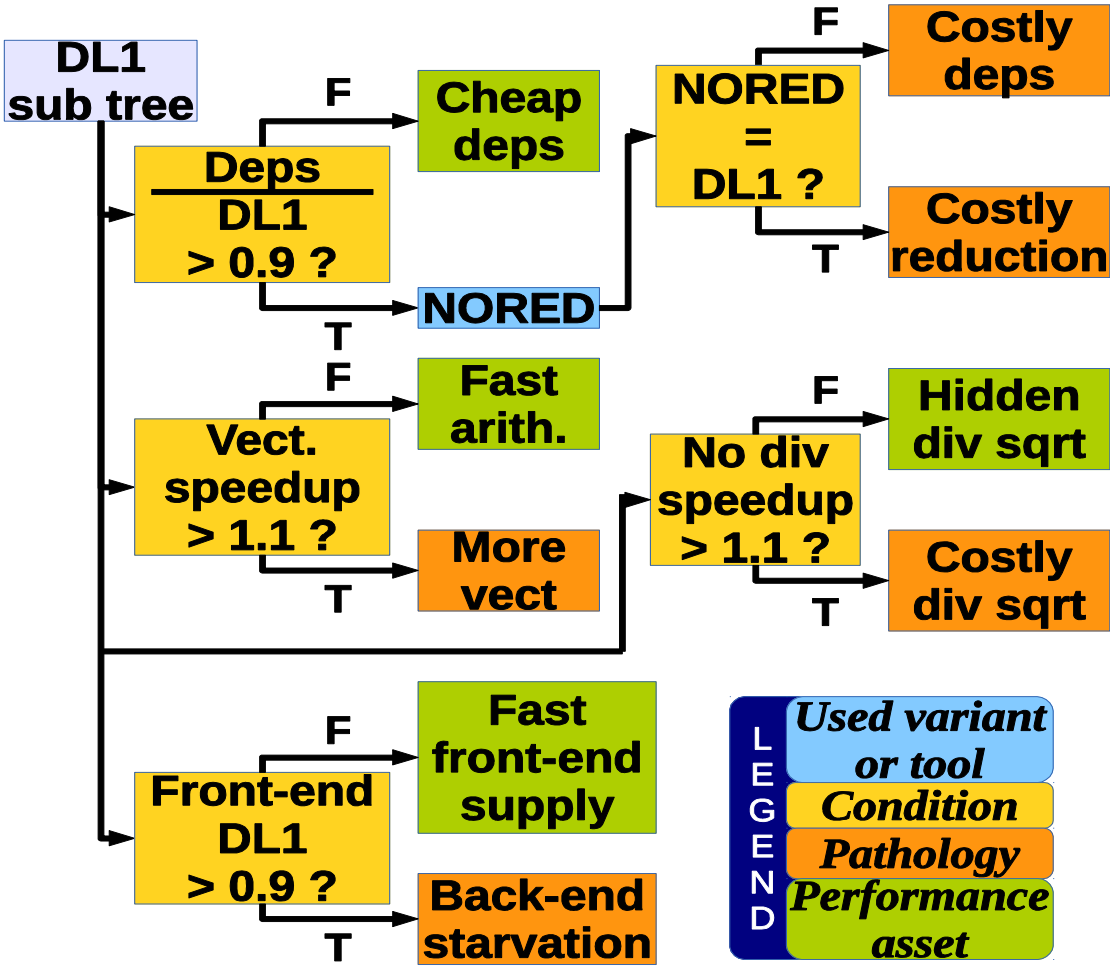



39

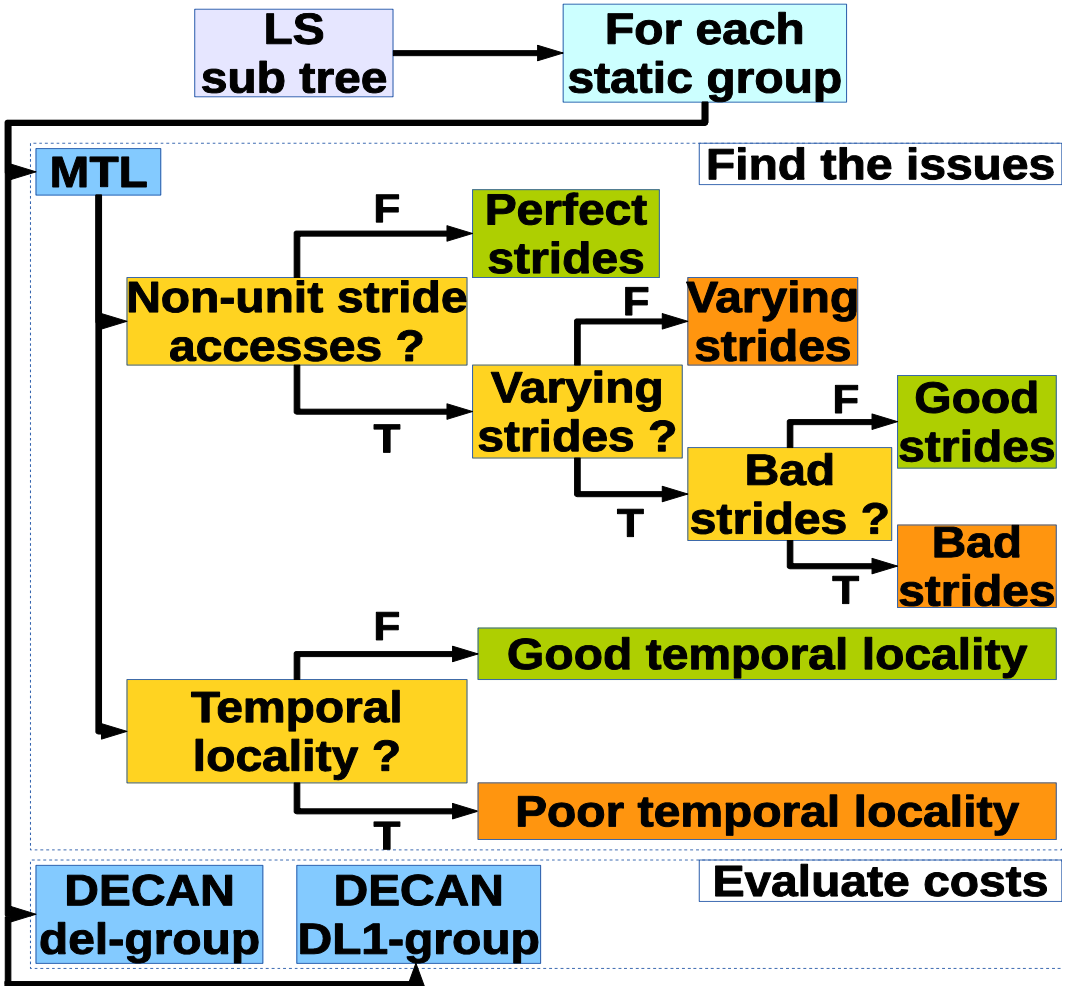
# Methodology *(main tree)*



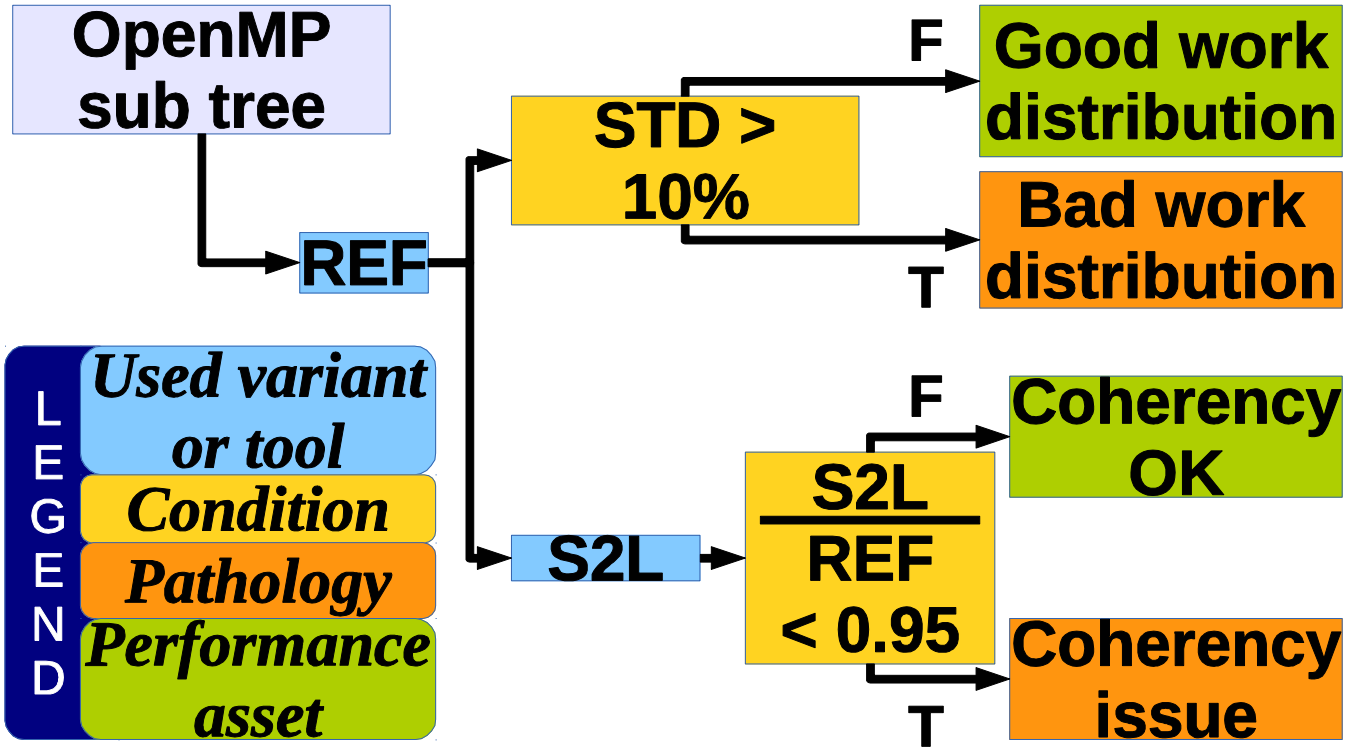
# Methodology *(CPU bound tree)*



# Methodology *(memory bound tree)*



# Methodology *(OpenMP tree)*






43