PAMDA: Performance Assessment using MAQAO toolset and Differential Analysis

Zakaria Bendifallah[1,2], William Jalby[1,2], José Noudohouenou[1,2], Emmanuel Oseret[1,2], Vincent Palomares[1,2] and Andres S. Charif Rubial [1,2]

[1] Exascale Computing Research, France
[2] University of Versailles Saint-Quentin-en-Yvelines, France
{firstname.lastname}@exascale-computing.eu

Abstract. Identifying performance bottlenecks in applications is crucial to improve their efficiency, but it might be difficult to precisely assess their impact on performance: in particular, two performance problems can interact making it difficult to isolate and therefore to correct them. We propose PAMDA, a methodology to single out performance problems through hierarchical bottlenecks detection. Important potential performance issues are classified in a 'Performance Breakdown Tree' which is used to drive our iterative analysis cycle, prioritizing the most relevant problems. Our system relies on MAQAO toolset and code's differential analysis. While MAQAO is a performance analysis and optimization tool suite, the differential analysis approach, which is implemented through DECAN tool, consists in quantifying performance changes when applying controlled transformations to the target code. Our focus will be on performance issues raised by processors and memory sub-systems in multicore architectures. We will demonstrate the approach on loops extracted from real life HPC applications.

1 Introduction

The recent progress of high performance architectures generate new challenges for performance evaluation tools: more complex processors (larger vectors, many-cores), more complex memory systems (multiple memory levels including NUMA, multiple level prefetch mechanisms), more complex systems (large increase in core counts up to several hundred of thousands now) are all key issues which need to be simultaneously optimized to get a decent performance level.

To work properly, all of these mechanisms require specific properties from the target code. For example, good exploitation of memory hierarchies relies on good spatial and temporal locality within the target code. The lack of such properties induces variable performance penalties: such combinations (mismatch between hardware and software) are denoted performance pathologies. Most of them have been identified (cf. Table 1) and efficient workarounds are well known. The current generation of performance tools (TAU [1], PerfExpert [2], VTune [3], Acumem [4], Scalasca [5], Vampir [6]) is excellent at detecting such pathologies although some are fairly specialized: for instance, Scalasca/Vampir mainly addresses MPI/OpenMP issues, requiring the combined use of several tools to get a global overview of all of the performance pathologies present in an application.

Most of the current tools do not provide any direct insight on the potential cost of a pathology. Furthermore, the user has no idea about what the potential benefit of optimizing his code to fix a given pathology is. These two points prevent him from focusing on the right issue. For example, let us consider a program containing two hot routines A and B, respectively consuming 40 % and 20 % of the total execution time. Let us further assume that the potential achievable performance gain on A is 10 % while on B it is up to 60 %. The overall performance impact on B is up to 60 % * 20 % = 12 % while on routine A, it is at best 10 % * 40 % = 4 %. As a consequence, it is preferable to focus on routine B. Additionally, the user has no clue of what the current performance level is, compared with the best one achievable, i.e. he may not know when optimizing is worth the investment.

In general, the situation is even worse since a simple loop may simultaneously exhibit several performance pathologies. In such cases, most of the tools cited above give no hint to the user of which ones are dominant and really worth fixing. For instance, a loop can suffer from both a high miss rate and the presence of costly Floating-Point (FP) operations such as div/sqrt: trying to improve the hit rate does not improve the performance if the dominant bottleneck is the div/sqrt operations.

In this paper, we present a coherent set of tools (MicroTools [7], CQA [8, 9, 10], DECAN [11], MTL [12]) to address this lack of user's guidance in the tedious and difficult task of program optimization. These tools are integrated in a unified environment (PAMDA) to help the user to quickly identify performance pathologies and to assess their cost and impact on the global performance. Depending upon which performance pathologies is to be fixed, different techniques (static analysis, value profiling, dynamic analysis) appear to be more appropriate and give a more accurate answer: for example, detecting a badly strided access is immediate through value tracing of array addresses while the same task is extremely tedious when only using static analysis or hardware counters. Anyway, such array access tracing should only be triggered when necessary due to its high cost. In this paper, we focus on providing performance insight at the core level and parallel OpenMP structures. Our analysis can be combined with MPI analysis provided by tools such as Scalasca, TAU or Vampir.

Through the integrated environment PAMDA, we aim at providing the following contributions:

- Getting a global hierarchical view of performance pathologies/bottlenecks.
- Getting an estimate of the impact of a given performance pathology taking into account all other present pathologies.
- Demonstrating that different specialized tools can be used for pathology detection and analysis.
- Performing a hierarchical exploration of bottlenecks according to their cost: the more precise but expensive tools are only used on specific well chosen cases.

Section 2 presents a motivating example in detail. Section 3 details the various key components of PAMDA while Section 4 describes the combined use of these

different tools. Section 5 describes some experimental use of PAMDA. Section 6 gives an overview of related works and the added value of the PAMDA system. Finally, Section 7 gives conclusions and future directions for improvement.

Table 1. A few typical performance pathologies.

Pathologies	**Issues**	**Workarounds**
ADD/MUL balance	ADD/MUL parallel execution (of fused multiply add unit) underused	Loop fusion, code rewriting e.g. Use distributivity
Non pipelined execution units	Presence of non pipelined instructions: div and 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 control flow graph in innermost loops	Prevents vectorization	Loop hoisting or code specialization
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
False sharing	Loss of bandwidth due to coherence traffic and higher latency access	Data padding or rearrange data structures
Cache leaking	Loss of bandwidth 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
High number of memory streams	Too many streams for hardware prefetcher or conflict miss issues	See conflict misses
Lack of loop unrolling	Significant loop overhead, lack of instruction-level parallelism	Try different unrolling factors, unroll and jam for loops nest, try classical affinities (compact, scatter, etc.)

2 Motivating Example

Figure 1 presents the source code of one of the hottest loops extracted from POLARIS(MD) [13]: a molecular dynamics application developed at CEA DSV. POLARIS(MD) is a multiscale code based on Newton equations: it has been successfully used to model Factor Xa involved in thrombosis.

This loop simultaneously presents a few interesting potential pathologies:

- Variable loop trip count.
- Fairly complex loop body which might lead to inefficient code generation by the compiler.
- Presence of div/sqrt operations.
- Strided and indirect access to arrays (scatter/gather type).
- Multiple simultaneous reduction operations leading to inter iteration dependencies.

All these pathologies can be directly identified by simple analysis of the source code. The major difficulty is to assess the cost of each of them and therefore to decide which should be worked on.

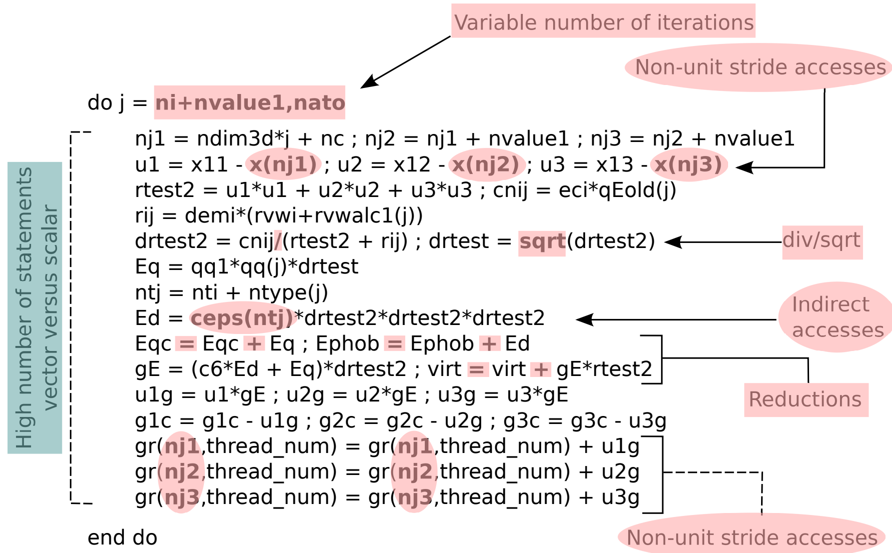

Fig. 1. A Fortran source code sample and its main performance pathologies highlighted in pink.

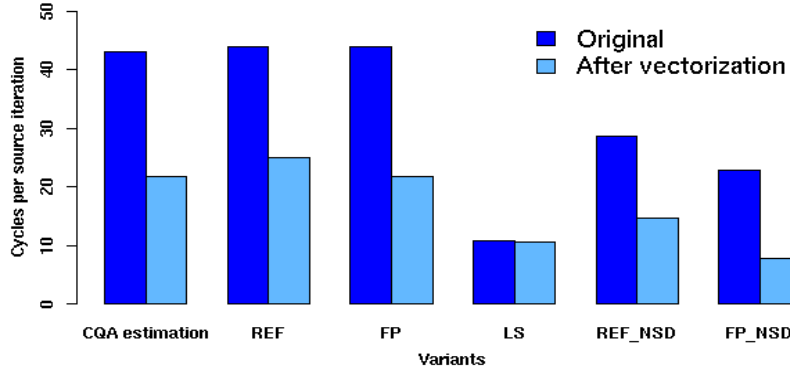

Fig. 2. Comparing static estimates obtained by CQA with dynamic measurements performed on different code variants generated by DECAN of both of the original and vectorized versions: **REF** is the reference binary loop (no binary modifications introduced by DECAN), **FP** (resp. **LS**) is the DECAN binary loop variant in which all of the Load/Store (resp. FP) instructions have been suppressed, **REF_NSD** (resp. **FP_NSD**) is the DECAN binary loop variant in which only FP div and sqrt instructions (resp. all of the Load/Store and FP div and sqrt instructions) have been suppressed. The y-axis represents the number of cycles per source iteration: lower is better.

A first value profiling of the loop iteration count reveals that the trip count is widely varying between 1 and 2000. However the amount of time spent in

the small (less than 150 iterations) loop trip count instances remains limited to less than 10 %. The remaining interval of loop trip counts is further divided into 10 deciles and one representative instance is selected for each of them. Further timings on analyzing loop trip count impact indicate that the average cost per iteration globally remains constant independently from the trip count. Therefore, the data size variation seems to have no impact on performance: the same optimization techniques should apply for instances having a loop trip count between 150 and 2000.

The static analyzer (see Figure 3) provides us with the following key information: in the original version, neither Load/Store (LS) operations nor FP ones are vectorized. It further indicates that due to the presence of div/sqrt operations, the FP operations are the main bottlenecks. It also points out that even if the FP operations were vectorized, the bottlenecks due to div/sqrt operations would remain. However this information has to be taken with caution since the static analyzer assumes that all data accesses are ideal, i.e. performed from L1.

Dynamic analysis using code variants generated by DECAN is presented in Figure 2. Initially, the original code (in dark blue bars) shows that FP operations (see FP versus LS DECAN variants) clearly are the dominating bottlenecks. Furthermore, the good match between CQA and REF clearly indicates that analysis made by CQA is valid and pertinent. Optimizing this loop is simply obtained by inserting the SIMD pragma '`!DEC$ SIMD`', which forces the compiler to vectorize FP operations. However, the compiler does not vectorize loads and stores due to the presence of strides and indirect access. Rerunning DECAN variants of this optimized version (see light blue bars in Figure 2) shows that even for this optimized version FP operations still remain the key bottleneck (comparison between LS and FP). Therefore, there is no point in optimizing data access, the only hope of optimization lies in improving div/sqrt operations: for example SP instead of DP. Unfortunately, such a change would alter the numerical stability of the code and cannot be used.

The major lesson to be drawn from this case study is that a combined use of CQA and DECAN allows us to quickly identify the optimization to be performed and also gives us a clear halt on tackling other pathologies without impacting overall performance.

3 Ingredients: Main Tool Set Components

Performance assessment issues require robust methodologies and tools. Therefore, in order to systematically provide programmers with a performance pathology hierarchy and its related costs, the current work considers two toolsets: MicroTools, for microbenchmarking the architecture, and the MAQAO [8, 9, 10] framework, which is a performance analysis and optimization tool suite.

MAQAO's goal is to analyze binary codes and to provide application developers with reports to optimize their code. The tool mixes both static (code quality evaluation) and dynamic (profiling, characterization) analyses based on the ability to reconstruct low level (basic blocks, instructions, etc.) and high level

structures such as functions and loops. Another MAQAO key feature is its extensibility. Users easily write plugins thanks to an embedded scripting language (Lua), which allows fast prototyping of new MAQAO-tools.

From MAQAO, PAMDA extensively uses three tools including the Code Quality Analyzer tool (CQA) exposed in section 3.2, the Differential Analysis framework (DECAN) presented in section 3.3, and finally the Memory Tracing Library (MTL) in section 3.4. We briefly present the main contribution of each of these tools to PAMDA and then describe their major characteristics.

3.1 MicroTools: Microbenchmarking the Architecture

Microbenchmarking [14, 15, 16] is an essential tool to investigate the real potential of a given architecture: more precisely, in PAMDA, microbenchmarking is first used to determine both FP units performance and achievable peak bandwidth of various hardware components such as cache/RAM levels, and second to estimate the potential cost of various pathologies (unaligned access, 4K aliasing, high miss rate, etc.).

For achieving these goals, PAMDA relies on `MicroTools` , consisting of two main components: `MicroCreator` tool automatically generates a set of benchmark programs, while `MicroLauncher` framework executes them in a stable and closed environment.

3.2 CQA: Code Quality Analyzer

In PAMDA, the CQA framework is used first for providing a performance target under ideal data access conditions (all operands are supposed to be in L1), second for providing a bottleneck hierarchy analysis between the various hardware components of the core (FP units, load/store ports, etc.) and third for detecting some performance pathologies (presence of inter iterations dependencies, div/sqrt operations) which are worth investigating via specialised DECAN variants. The ideal assumption (all operands in L1) is essential for CPU bound codes such as the POLARIS(MD) loop studied in the previous section. For memory bound loops, it needs to be complemented with a dynamic analysis.

CQA is a static analysis tool directly dealing with binary code. It extracts key characteristics, and detects potential inefficiencies. It provides users with general code metrics such as details on basic loop characteristics, the number of instructions, μ`ops`, and used XMM/YMM vector registers. CQA also allows users to obtain more in-depth information on the loop execution on the target architecture. For example, the tool provides a reliable front-end pipeline execution report, which is an estimated number of cycles spent during each front-end pipeline stage. The tool gives the same type of report for the back-end. Finally, CQA provides a cycle estimate of loop body performance under ideal conditions: all operands in L1, no branches and infinite loop count (steady state behavior).

CQA is able to report both low and high level metrics/reports (figure 3). For example, when a loop is not fully vectorized, the high level report provides a potential speedup (if all instructions were vectorized) and corresponding hints

(compiler flags and source transformations). For the same loop, some low level metrics/reports show the breakdown of vectorization ratios per instruction type (loads, stores, ADDs, etc.) giving the user a more in-depth view of the issue.

CQA supports Intel 64 micro-architectures from Core 2 to Ivy Bridge.

```
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%

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

Fig. 3. CQA output.

3.3 DECAN: Differential Analysis

In PAMDA, DECAN is used for quantitatively assessing performance pathologies impact. The general idea is fairly simple: a given pathology is associated with the presence of a given subset of instructions, for example div/sqrt operations, then DECAN generate a binary version of the loop in which the corresponding instructions are deleted or properly modified. This altered binary is measured and compared with the original unmodified version.

The resulting binary does not in general preserve semantics, i.e. numerical values generated with DECAN variants are not identical to the original ones. For our performance analysis objective, this is not a critical issue but for the subsequent program execution, control behavior might be altered. To avoid such problems, the original loop is systematically replayed after the execution of the modified binary in order to restore correct memory values.

DECAN starts by using static analysis on the target loop produced by CQA. The goal is to select instruction subsets to be transformed, as the selection process is driven by the desired type of behavior to highlight. Afterward, instructions are carefully transformed in a manner that minimizes unwanted side effects that may disturb the observations, such as changes in the code layout and instruction dependencies. It also inserts some monitoring probes to be able to accurately compare the modified part of the code with the original one. Also, and as stated earlier, DECAN is built on top of the MAQAO framework, hence, it uses the MAQAO disassembler/patcher to forward modifications on the instructions.

Using DECAN's features, PAMDA generates altered binaries, thereby splitting performance problems between CPU, memory, and OpenMP issues. Table 2 presents a range of loop variants used within the methodology discussed in Section 4.

Table 2. DECAN variants and transformations.

Variant	Type of SSE/AVX instructions involved	Transformation
LS	All arithmetic instructions	Instruction deleted
FP	All memory instructions	Instruction deleted
DL1	All memory instructions	Instruction operands modified to target a unique address
NODIV	All division instructions	Instruction operands modified to target a unique addresses
NORED	All reduction instructions	Instruction deleted
S2L	All store instructions	Converted into load instructions
NO_STORE	All store instructions	Instructions are deleted

3.4 MTL: Memory Tracing Library

Within PAMDA, MTL provides specific analysis of pathologies related to data access patterns in particular stride values, alignment characteristics, data sharing issues in multi-threaded codes, etc. MTL works by tracing addresses and by generating compact representations of data access patterns. MTL is not limited to innermost loops but directly deals with multiple nested loops, allowing to detect more subtle pathologies: for example, row major instead of column major accesses for a Fortran array (stored column wise) are automatically detected. To perform these analysis, MTL uses the MAQAO Instrumentation Language (MIL) [17]. This language makes the development of program analysis tools based on static binary instrumentation easier. In fact, MIL is a specific language for object-oriented and event-directed domains to perform binary instrumentation at a high level of abstraction using structural objects (functions, loops, etc.), events, filters, and probes.

4 Recipe: PAMDA Tool Chain

Individual tools are the building blocks that PAMDA glue together through a set of scripts (cf all the diagrams). These scripts are under development but most of the principles have been already evaluated. Figure 4 presents PAMDA overall organization, which includes application profiling, cost analysis, structural checks, CPU and memory subsystems evaluation, and finally OpenMP evaluation for parallel applications. The current section describes PAMDA's components.

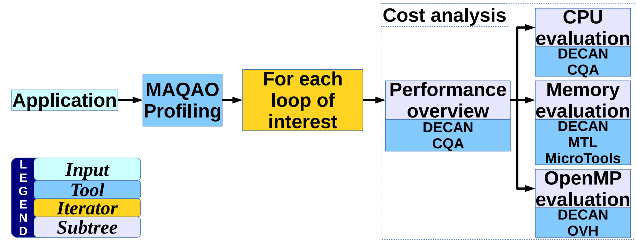

Fig. 4. PAMDA overview.

4.1 Hotspot identification

To limit the processing cost, we focus on the most time consuming portions of the code. Our target loops are defined as the loops with a cumulated execution time exceeding 80% of the total execution time. It should be noted that with such an aggregated measurement, we can end up with a large number of loops with small individual contributions. Such target loops are identified using `MAQAO sampling`.

4.2 Performance overview

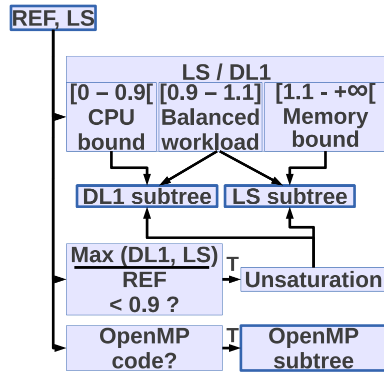

Fig. 5. Performance investigation overview. '?' represents a condition and 'T' means the condition is True, otherwise it is False ('F').

The PAMDA approach divides performance bottlenecks into two main categories (Figure 5): memory subsystem and CPU. Then, their respective contribution to the overall execution time is quantified using DECAN transformations `LS` (assessing memory subsystem performance) and `DL1` (assessing CPU subsystem performance). The ratio of these contributions reveals whether the loop is memory or/and CPU bound. Ideally, pipeline and out of order mechanisms insure that cycles spent for memory accesses and for arithmetic operations perfectly overlap: as a result, the time taken by `REF` should be the maximum time taken either by `LS` or `DL1`. In such a case, only the slower component needs optimizing. If the time taken by `LS` and `DL1` is similar, the workload is said to be balanced: optimizing both components is necessary to improve the loop's performance. Finally, when cycles taken by the memory and CPU components are poorly covered by one another (`unsaturation`), optimizing either of them can be sufficient to gain overall performance.

4.3 Loop structure check

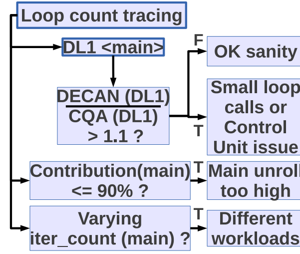

Fig. 6. Detecting structural issues. 'Iter_count' illustrates iteration counts from the main loop.

Loop structure issues can be detrimental to performance, and may be detected using DECAN's loop trip counting feature. Indeed, in the case of unrolling or vectorization, peel and tail scalar codes may have to be generated to cover for remaining iterations. If too much time is spent in these peel and tail codes, this might indicate the unroll factor is too high with respect to the source loop iteration count. To detect such cases, loop trip counts for each version (peel/tail/main) are determined, and we check whether the main loop is processing at least 90 % of the source code iterations.

In some cases, the number of iterations per loop instance may not be large enough to fully benefit from unrolling or vectorization. This is easily highlighted by comparing the dynamic execution time of the DL1 DECAN variant with the CQA estimate, as the latter assume an infinite trip count. The difficulty to optimize such loops is exacerbated when the loop trip counts are not constant.

4.4 CPU evaluation

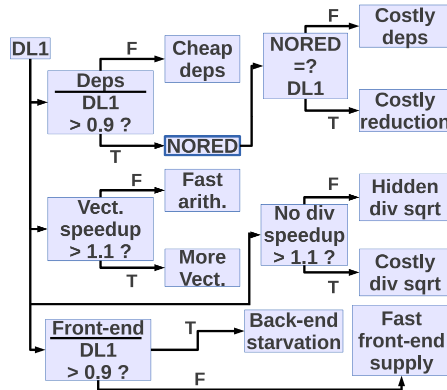

Fig. 7. DL1 subtree: CPU performance evaluation. Except DL1 and NORED, others metrics used by conditioners are extracted from CQA static analysis.

Besides data accesses, CPU performance may be limited by other pathologies such as long dependency chains (`deps`), reductions (`RED`), scalar instructions or long latency floating point operations (`div`): these pathologies can be detected through the combined use of CQA and DECAN (Figure 7). The front-end can also slow down the execution by failing to provide the back-end with micro-operations at a sufficient rate. Comparing their contribution to L1 performance (`DL1`) is a cost-effective way to identify such problems. Finally, CQA can provide us with estimations of the effect of vectorizing a loop. We precisely quantify CPU related issues, enabling us to reliably assess potential for optimizations such as getting rid of divisions, suppressing dependencies or vectorizing. This information can guide the user's optimization decisions.

4.5 Bandwidth measurement

Data access rates from different cache levels / RAM highly depend on several factors, such as the instructions used or the access pattern.

To this end, we generate microkernels loading data in an ideal stream case, testing different configurations for load operations, with or without various software prefetch instructions, and/or splitting the accessed data in streams accessed in parallel. We also force misaligned addressing for `vmovups` and `movups`. Finally, we use `Microlaunch` to run these benchmarks for each level of the memory hierarchy.

Table 3. Various vector/scalar load bandwidths estimation in bytes per cycle for each memory level (Sandy Bridge E5-2680).

	Instruction	L1	L2	L3	RAM
AVX	vmovaps	31.74	15.05	10.81	5.10
	vmovups	31.73	14.96	10.81	5.10
SSE	movaps	30.72	18.16	10.80	5.14
	movups	29.53	17.07	10.79	5.23
	movsd	15.67	11.55	10.61	5.36
	movss	7.91	6.65	6.39	4.97

On our target architecture, 128-bit SSE load instructions could roughly achieve the same bandwidth as 256-bit AVX (Table 3) throughout the whole memory hierarchy. Except for `movss`, all instructions could attain similar bandwidths in L3 and RAM: only the type of instruction really matters for data accesses from L1 or L2, and data alignment is not as relevant as it once was.

4.6 Memory evaluation

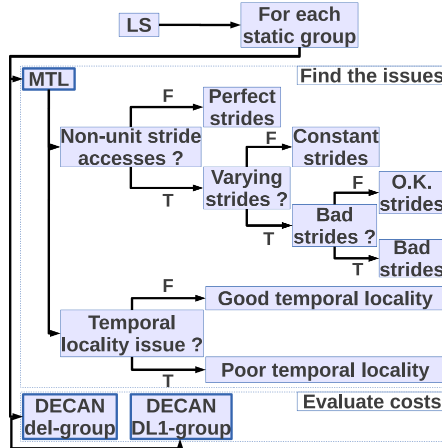

Fig. 8. LS subtree: Memory performance evaluation.

Memory performance can be quite complex to evaluate. We use MTL to find the different access patterns and strides for each memory group (as defined by the grouping analysis [11]). Memory accesses typically are more efficient when targeting contiguous bytes, while discontiguous accesses reduce the spatial locality of data. The worst case scenario is having large and unpredictable strides, as hardware prefetchers may not be able to function properly. MTL also provides the data reuse distance, allowing the temporal locality evaluation of groups.

Once potential performance caveats are identified, we can use DECAN transformation `del-group` to single out offending groups and quantify their contribution to the `LS` variant global time.

Comparing the bandwidth measured for each group with the bandwidth obtained in ideal conditions in the bandwidth measurement phase may then provide us with an upper limit on achievable performance.

4.7 OpenMP evaluation

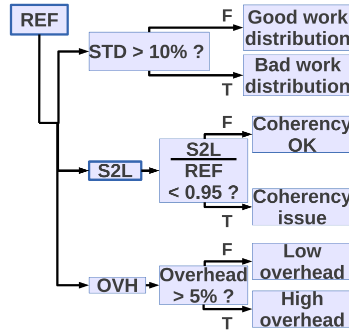

Fig. 9. OpenMP performance tree: STD represents the standard deviation between threads while the OVH branch stands for OpenMP Overhead evaluation.

Some issues are specific to parallel programs using OpenMP (Figure 9). The standard deviation (`STD`) of the execution time for each thread points out workload imbalances. It is particularly important that no thread takes significantly longer than others to compute its working set, as loop barriers may then highly penalizing stalls. Another issue is excessive cache coherency traffic generated by store operations on shared data. Transformation `S2L` converts all stores to loads: we can quantify coherency penalties by comparing `S2L` with `REF`. Furthermore, the OpenMP Overhead (OVH) module of MAQAO is able to measure the portion of time spent in OpenMP routines, providing an OpenMP overhead metric.

5 Experimental results

We applied our methodology on two scientific applications: PN and RTM. The analysis processes and test results are presented below.

5.1 PN

PN is an OpenMP/MPI kernel used at CEA (French Department of Energy). Hot loops are memory bound and are ideal to stress tools dedicated to memory optimizations.

All tests are performed on a two-socket Sandy-Bridge machine, composed of two Intel E5-2680 processors with 8 physical cores each.

The profiling done on the initial MPI version of the code presents four loops consuming more than 8% of the global execution time each. Because of a lack of space, we only study the first one, but the three other loops have a similar behavior.

According to the methodology, the next step consists in gaining more insight on the loop characteristics through *performance overview*, hence, the `LS` and `DL1` DECAN variants are used. The corresponding results shown in Figure 10

indicate a strong domination of data accesses, with the `LS` curve being well over the `DL1` curve and matching the `REF` one. Consequently, the investigation follows the `LS` subtree.

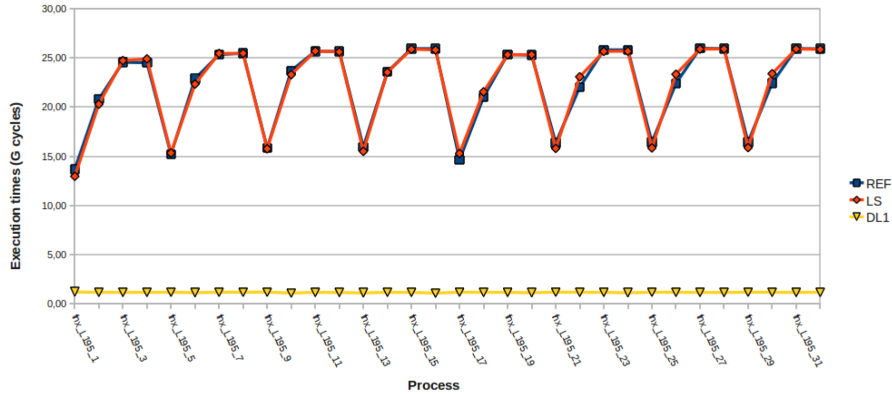

Fig. 10. Streams analysis on PN. The REF curve correspond to performance of the original code. The LS (resp. DL1) curve corresponds to the DECAN variant where all FP instructions have been suppressed (resp. all data accesses are forced to come out of L1).

In order to get more information on data accesses, we use MTL. Six instruction groups are detected but only three of them contain relevant SSE instructions dealing with FP arrays. Experimental results in Table 4 illustrates MTL output, which uses i1, i2, and i3 to represent loop indices leading to conclude the considered piece of code contains at least a triple-nested loop. Table 4 analysis indicate a simple access pattern for group G1 (stride 1) and, for groups G6 and G5, more complicated patterns which need to be optimized. As a result, in this step we are able to characterize our memory accesses with precision. Though, it leaves us with two accesses and no possibility to know which one is the most important. At this point, we return to our notion of ROI provided through Differential analysis and apply the DECAN `del-group` transformation for each of the three selected groups. The `del-group` results shown in Figure 11 clearly indicate that G6 is the most costly group by far: it should be our first optimization target.

Table 4. PN MTL results for the three most relevant instruction groups. i1, i2, and i3 represent loop indices.

Group	Instructions	Pattern
G1	Load (Double)	8*i1
G6	Load (Double)	8*i1+217600*i2+1088*i3
G5	Store (Double)	8*i1+218688*i2+1088*i3

With the finding of the delinquent instruction group, the analysis phase comes to its end. The next logical step is to try and optimize the targeted memory access. Fortunately, the information given by MTL reveals an interesting pathology. The access pattern of the instruction of interest has a big stride in the innermost loop (1088*i3) and a small one in the outermost loop (8*i1). In order to diminish the access penalty we perform loop interchange between the two loops, which results in a considerable performance gain at the loop level with a speedup of *7.7*x and consequently a speedup of *1.4*x on the overall performance of the application.

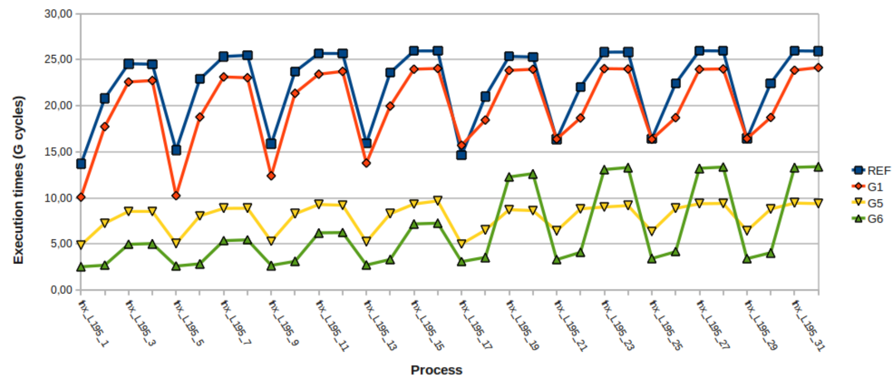

Fig. 11. Group cost analysis on PN. Each group curve corresponds to performance of the loop while the target group is deleted. The original code performance (REF) is used as reference.

5.2 RTM

Reverse Time Migration (RTM) [18] is a standard algorithm used for geophysical prospection. The code used in this study is an industrial implementation of the RTM algorithm by the oil & gas company TOTAL.

Our RTM code operates on a regular 3D grid. More than 90% of the application execution time is spent in two functions, `Inner` and `Damping`, which execute similar codes on two different parts of the domain: `Inner` is devoted to the core of the domain while `Damping` is used on the skin of the domain. Standard domain decomposition techniques are used to spread the workload on multicore target machines. Since the grid is uniform, load balancing can be easily tuned by using rectangular sub-domain decomposition and by properly adjusting the sub-domain size.

All experiments are done on a single socket machine, which contains a quad-core Intel Xeon E3-1240 processor with a cache hierarchy of 32KB (L1), 256KB (L2) and 8MB (shared L3).

Step 1: The original version of the code is provided with a default non-optimized blocking. The first analysis on the OpenMP subtree reveals an imbalanced work sharing. A second analysis done at the level of the *performance overview* subtree shows that the code is highly bounded by memory operations. In order to fix this, we focus on the blocking strategy. As a result it turns out that the default block size is responsible for both the load imbalance between threads and the bad memory behavior. We can then select a strategy which provides a good balance at the work sharing level as well as a good trade-off between the LS and FP streams. However, we note that, to obtain an optimal strategy, a more dedicated tool should be used.

Step 2: The second step of the analysis consists of going further in the OpenMP subtree and checking how the RTM code performs in term of coherency. As explained earlier, the structure of the code induces a non-negligible coherency traffic. Figure 12 shows experimental results after applying the S2L transformation on RTM. While the x-axis details loops respectively identified from `Inner` and `Damping` functions, the y-axis represents speedups over the original loops. The results indicate a negligible gain due to canceling potential coherency modifiers and a minimal gain, observed on two loops, due to a complete deletion of the stores. Consequently, we can conclude that maintaining the overall coherency state remains negligible, therefore, there would be no point in going further in this direction.

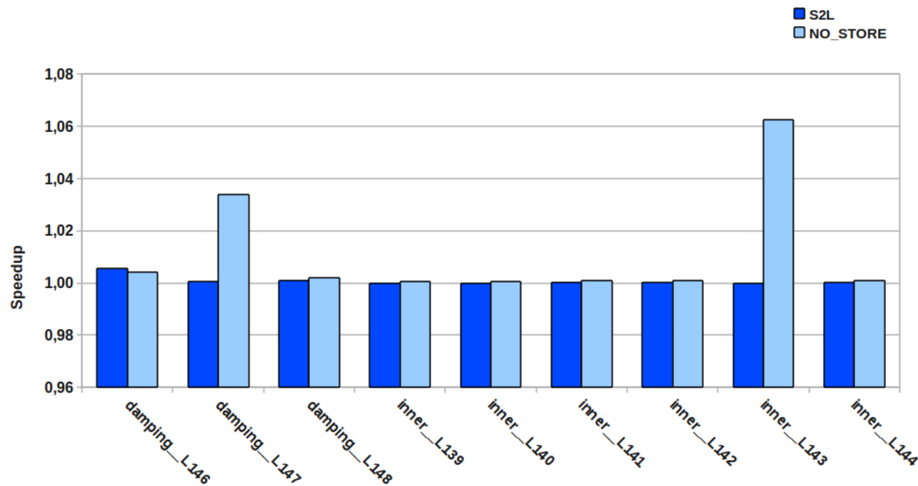

Fig. 12. Evaluation of the cost of cache coherence protocol. The S2L variants show similar performance as their corresponding reference versions. The NO_STORE variants show similar performances also, except for two loops which present a relatively non negligible store cost.

6 Related Work

Improving an application's efficiency requires identifying performance problems through measurement and analysis but assessing bottlenecks impact on performance is much harder. To achieve that, most researchers consider a qualitative approach. TAU [1] represents a parallel performance system that addresses diverse requirements for performance observation and analysis. Although performance evaluation issues require robust methodologies and tools, TAU only offers support to the performance analysis in various ways, including instrumentation, profiling and trace measurements.

Tools such as Intel VTune [3], GNU profiler (Gprof) [19], Oprofile [20], MemSpy [21], VAMPIR [6], and Scalasca [5] provide considerable insight on the application's profile. In term of methodology Scalasca, for instance, proposes an incremental performance-analysis procedure that integrates runtime summaries based on event tracing. While these tools help hardware and software engineers find performance pathologies, significant manual performance tuning remain for software improvements, for example, selecting instructions in particular part of a program.

PerfExpert [2], HPCToolkit [22], and AutoSCOPE [23] pinpoint performance bottlenecks using performance monitoring events. Furthermore, while PerfExpert suggests performance optimizations, AutoSCOPE extends PerfExpert by automatically determining appropriate source-code optimizations and compiler flags. Contrary to PAMDA, the considered tools do not provide a methodology presenting the cost related to the identified bottleneck. ThreadSpotter also helps a programmer by presenting a list of high level advice without addressing return on investment issues: what to do in case of multiple bottlenecks? How much do bottlenecks cost?

Interestingly in [24, 25], the authors present an automated system that fingerprints the pathological patterns of the hardware performance events and identifies the pathologies in applications, allowing programmers to reap the architectural insights. The proposed technique is close to the current work and includes pathology description through microbenchmarks as well as pathology identification using a decision tree. However, in order to evaluate usual performance pathologies, PAMDA additionally integrates pathology cost analysis.

The above survey indicates that performance evaluation requires a robust methodology, but traditional methods do not help much with coping with the overall hardware complexity and with guiding the optimization effort. Also, previous works focus on performance bottleneck identification providing optimization advice without providing potential gains. The previous factors motivate to consider PAMDA as the only methodology combining both qualitative and quantitative approaches to drive the optimization process.

7 Conclusion and Future Work

Application performance analysis is a constantly evolving art. The rapid changes in the hardware mixed with new coding paradigms force analysis tools to handle

as many pathologies as possible. This can only be achieved at the expense of usability. At the end, application developers work with extremely powerful tools but they have to face significant differences and difficulties to use them.

This paper illustrates the usefulness of performance assessment combining static analysis, value profiling and dynamic analysis. The proposed tool chain, PAMDA, helps the user to quickly identify performance pathologies and assess their cost and impact on the global performance.

The goal in using PAMDA is to make sure that the right effort is spent at each step of the analysis and on the right part of the code. Furthermore, we try to create some synergy between different tools by combining them in a unified environment. We provide some case studies to illustrate the overall analysis and optimization process. Experimental results clearly demonstrate PAMDA's benefits.

Obviously, the provided methodology is far from being finished. Our constant challenge is to keep improving it as well as working towards full automation. We also aim to enlarge it for other kind of paradigms through the integration of analyses provided by complementary tools such as Scalasca, Vampir and TAU. Additionally, refining optimization investigations is crucial in order to make it more user-friendly.

8 Acknowledgments

We would like to thank Michel Masella for the access to his POLARIS(MD) code and Henri Calandra and Asma Farjallah for the access to the RTM code.
This work has been carried out by the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, UVSQ, and by the PRiSM laboratory, thanks to the support of the French Ministry for Economy, Industry, and Employment throught the PERFCLOUD project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

References

[1] Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. **20**(2) (May 2006) 287–311
[2] Burtscher, M., Kim, B.D., Diamond, J.R., McCalpin, J.D., Koesterke, L., Browne, J.C.: PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In: SC, IEEE (2010) 1–11
[3] Intel: Intel VTune Amplifier XE. www.intel.com/software/products/vtune (2013)
[4] Acumem: Acumem ThreadSpotter. `http://www.roguewave.com/products/threadspotter.aspx`
[5] Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahàm, E., Becker, D., Mohr, B., Jlich, F.: The SCALASCA performance toolset architecture. In: STHEC. (2008)
[6] Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer **12** (1996) 69–80

[7] Beyler, J.C., Triquenaux, N., Palomares, V., Chabane, F., Fighiera, T., Halimi, J.P., Jalby, W.: MicroTools: Automating Program Generation and Performance Measurement. In: ICPPW, 2012, IEEE (2012) 424–433
[8] Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance Tuning of x86 OpenMP Codes with MAQAO. In: Parallel Tools Workshop, Dresden, Germany, Springer-Verlag (sep 2009)
[9] Koliai, S., Zuckerman, S., Oseret, E., Ivascot, M., Moseley, T., Quang, D., Jalby, W.: A Balanced Approach to Application Performance Tuning. In: LCPC. (2009) 111–125
[10] MAQAO: MAQAO Project. http://www.maqao.org (2013)
[11] Koliaï, S., Bendifallah, Z., Tribalat, M., Valensi, C., Acquaviva, J.T., Jalby, W.: Quantifying performance bottleneck cost through differential analysis. In: 27th ICS, New York, NY, USA, ACM (2013) 263–272
[12] Charif-Rubial, A.S.: On code performance analysis and optimisation for multicore architectures. PhD thesis (October 2012)
[13] Real, F., Trumm, M., Vallet, V., Schimmelpfennig, B., Masella, M., Flament, J.P.: Quantum Chemical and Molecular Dynamics Study of the Coordination of Th(IV) in Aqueous Solvent. J. Phys. Chem. B **114**(48) (2010) 15913–15924
[14] Staelin, C., packard Laboratories, H.: lmbench: Portable Tools for Performance Analysis. In: USENIX Annual Technical Conference. (1996) 279–294
[15] Liu, J., Yu, W., Wu, J., Buntinas, D., Kini, S., K, D., Wyckoff, P.: Microbenchmark performance comparison of high-speed cluster interconnects. IEEE Micro (2004)
[16] Alam, S.R., Barrett, R.F., Kuehn, J.A., Roth, P.C., Vetter, J.S.: Characterization of scientific workloads on systems with multi-core processors. In: IISWC. (2006) 225–236
[17] Charif-Rubial, A.S., Barthou, D., Valensi, C., Shende, S.S., Malony, A.D., William Jalby, i.p.: MIL: A language to build program analysis tools through static binary instrumentation. In: HiPC'13, Hyderabad, India (December 2013)
[18] Baysal, E., Kosloff, D., Sherwood, J.: Rerverse Time Migration:Geophysics (1983)
[19] Gprof: The GNU Profiler. http://sourceware.org/binutils/docs-2.18/gprof/index.html (2013)
[20] Levon, J., Elie, P.: OProfile: A System Profiler For Linux. http://oprofile.sourceforge.net (2013)
[21] Martonosi, M., Gupta, A., Anderson, T.: MemSpy: Analyzing Memory System Bottlenecks in Programs. In: Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems. (1992) 1–12
[22] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. Concurr. Comput. : Pract. Exper. **22**(6) (April 2010) 685–701
[23] Sopeju, O., Burtscher, M., Rane, A., Browne, J.: AutoSCOPE: Automatic Suggestions for Code Optimizations using PerfExpert. In: 2011 ICPDPTA. (July 2011) 19–25
[24] Yoo, W., Larson, K., Baugh, L., Kim, S., Campbell, R.H.: ADP: automated diagnosis of performance pathologies using hardware events. In Harrison, P.G., Arlitt, M.F., Casale, G., eds.: SIGMETRICS, ACM (2012) 283–294
[25] Yoo, W., Larson, K., Kim, S., Ahn, W., Campbell, R.H., Baugh, L.: Automated Fingerprinting of Performance Pathologies Using Performance Monitoring Units (PMUs). In: 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar '11), Berkeley, CA, USENIX, USENIX (05/2011 2011)