CQA: A Code Quality Analyzer tool at binary level

Andres S. Charif-Rubial, Emmanuel Oseret, José Noudohouenou, William Jalby
Exascale Computing Research Laboratory, FR
Email: {achar,eoseret,jose.noudohouenou,william.jalby}@exascale-computing.eu
Ghislain Lartigue , Normandie Universite, FR
Email: ghislain.lartigue@coria.fr

***Abstract*—Most of today's performance analysis tools are focused on issues occurring at multi-core and communication level. However there are several reasons why an application may not correctly behave in terms of performance at the core level. For a significant part, loops in industrial applications are limited by the quality of the code generated by the compiler and do not always fully benefit from the available computing power of recent processors. For instance, when the compiler is not able to vectorize loops, up to a 8x factor can be lost. It is essential to first validate the core level performance before focusing on higher level issues.**

This paper presents the CQA tool, a loop-centric code quality analyzer based on a simplified unicore architecture performance modeling and on quality metrics. The tool analyzes the quality of the code generated by the compiler. It provides high level metrics along with human understandable reports that relates to source code. Our performance model assumes that all data are resident in the first level cache. It provides architectural bottlenecks and an estimation of the number of cycles spent in each iteration of a given innermost loop. Our modeling and analyses are statically done and requires no execution or recompilation of the application. We show practical examples of situations where our tool is able to provide very valuable information leading to a performance gain.

***Keywords*-performance evaluation, performance modeling; vectorization, static analysis;**

I. Introduction

Most of today's performance analysis tools [1], [3], [4], [6], [7], [13], [18], [21], [22], [24] are focused on issues occurring at multicore (shared memory resources) and communication level (message passing). However there are several reasons why an application may not correctly behave in terms of performance at the core level. For a significant part, loops in industrial applications are limited by the quality of the code generated by the compiler and do not always fully benefit from the available computing power of recent processors.

Analyzing the quality of the code generated by the compiler is the cornerstone to ensure the highest execution efficiency at the core level on a target architecture. The main goal is to evaluate the quality of the exploitation of each feature of a hardware architecture. On Intel 64 and Xeon Phi architectures, the aim of code quality assessment is to leverage the data parallelism (SIMD) and track down poor code patterns. The CQA contains two fundamental aspects. First modeling the behavior of the execution pipeline and then defining a set of metrics that can help us determine which architectural speedup opportunities could be worth. Even if only Intel 64 and Xeon Phi architectures are covered, our analysis is general enough to be applied to other architectures.

CQA does not cover memory issues. It must be used when the remaining bottlenecks are computational, that is to say when data is in the first level cache. It can also be used when vectorization is the main goal for instance when nothing else can be done to solve memory issues. The cycles estimation provides a lower bound which can be compared with the real execution time (i.e. from a profiler) to determine how far a given loop is from the optimum.

It is essential to first validate the core level performance before focusing on higher level issues. After computational bottlenecks have been removed, most performance gains should benefit all the involved cores. Scalability issues have then to be fixed to obtain a good overall efficiency. For instance, when the compiler is not able to vectorize the loops, up to an 8x factor can be lost on recent processors (e.g. Intel Haswell). Another example is being bounded by the weight of expensive instructions. Figure 1 depicts a simple OpenMP reduction

```
void red (int n, float *sum, float a[n], float b[n],
          int inda[n], int indb[n])
{
  int i;
  float _sum = 0.0;

  #pragma omp parallel for reduction(+:_sum)
  for (i=0; i<n; i++)
    _sum += sqrt (a[inda[i]] / b[indb[i]]);

  *sum = _sum;
}
```

Fig. 1. OpenMP reduction example in C language

code snippet. The code is compiled with the *gcc* compiler. It provides the highest optimization level $-O3$. Let us consider that this loop is really time consuming in a given application. A typical profiling phase will point out this loop. The first and intuitive idea in terms of optimization strategy would be to remove indirect access because the application will be executed on multicore shared memory systems. However the real bottleneck in this example is the square root and the division which together have a higher cost.

CQA presented in this paper proposes the following contributions:

- Code quality metrics such as vectorization ratio, number of arithmetic instructions, number of expensive instructions, etc.

- Supporting all Intel 64 and Xeon Phi micro-architectures starting back from Core 2 Duo to Haswell.
- A qualitative analysis of codes generated by the compiler and based on our performance model and architecture metrics.
- A static evaluation requiring no application execution. It only requires analyzing a binary. Results of this analysis are formatted as human understandable reports that can lead to performance gains when applying suggested hints.

The paper is organized as follows: section II depicts the place of CQA in the current performance evaluation tools landscape and also its integration into the MAQAO [23] tool. Section III describes our performance models for Intel 64 and Xeon Phi architectures. Then, section IV presents the architectural metrics that can be extracted from assembly code analysis along with the available speedup opportunities. Section V discusses the design decisions, implementation issues and limitations of the tool. Finally, section VI presents a set of case studies where CQA is able to provide very valuable information leading to a performance gain.

II. Context and Methodology

Before diving into a detailed description of the tool an overview of where it fits in the big picture of performance evaluation tools is provided.

A. Context

Performance issues slowing down applications can be the consequence of multiple factors like computation, memory or network issues (bottlenecks). There is no magic tool to find out all issues an application can suffer from. Generally multiple tools are used to target one category of issues. Nowadays the majority of tools mainly focus on communication issues due to scaling matters and in a lesser extend, on memory issues. Computation issues are considered as less important as memory issues. There are still four main scenarios where they have to be tackled:

- when computation bottlenecks are more important than memory usage
- when the code is not or partially vectorized
- when the compiler did generate very poor quality code patterns
- when memory issues have been fixed but the code does not run at full steam at the core level

The aim of CQA may seem pretty narrow but these scenarios may be more frequent than one can think. We show in this paper that such a tool can help fixing performance issues.

B. MAQAO and the Methodology

To tackle the emerging challenge of performance analysis on complex many-core systems and hardware accelerators the University of Versailles has been developing a performance analysis and optimization toolchain called MAQAO: the **M**odular **A**ssembly **Q**uality **A**nalyzer and **O**ptimizer (**MAQAO**). The main goal of MAQAO is to analyse binary codes and provide application developers with reports in order

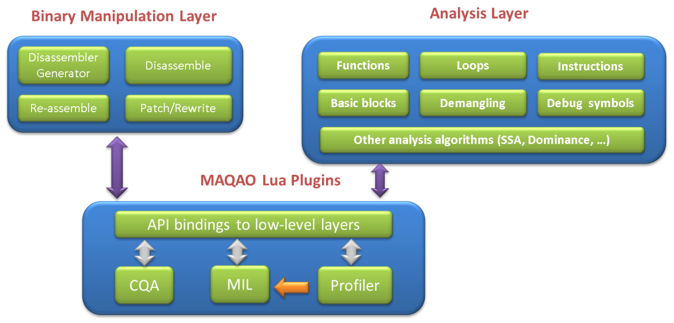

Fig. 2. MAQAO Framework overview

to help them optimize their code. Another main key feature of MAQAO is its extensibily. Users can easily write their own plugins thanks to an embedded scripting language (Lua) and an instrumentation language (MIL). It allows fast prototyping of new MAQAO-tools.

Figure 2 depicts the layers of the MAQAO framework.

CQA is typically used after a profiling pass performed by the MAQAO performance evaluation tool which provides function and loop hotspots. CQA will only run on the specified innermost loops, where usually most of the time is spent. For applications containing few hot inner loops, it is sometimes possible to extend an optimization from one hot loop to many others. As described later on the experiments section, optimizing the 5-10 hottest YALES2 loops benefited many other loops by:

- reporting to the Intel compiler group an issue preventing the compiler to optimize double indirections arrays (Fortran 90: a%b%array)
- replacing pointers to shared arrays with *allocatable* references

Optionally, the architecture of a target machine can be provided. The default one is the current machine the tool is ran

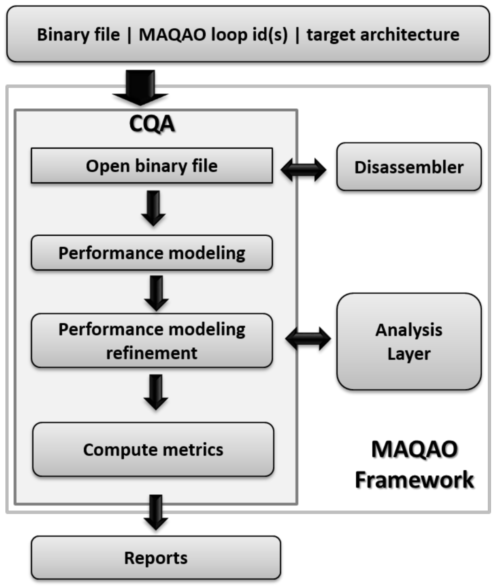

Fig. 3. MAQAO framework parts used by CQA

on. For more flexibility, it is also possible to specify functions instead of loops. In that mode, all innermost loops in functions with a name matches provided *regexps* are analyzed.

C. Integration of CQA into MAQAO

CQA was built as a MAQAO module using the MAQAO Lua API. It utilizes the *binary manipulation layer* and *Analysis layer* of the framework. The *binary manipulation layer* is used to open a given binary, to disassemble it and to extract source correlation information when available (e.g. source lines). The *Analysis layer* provides the following information for a given loop:

- data dependency graph (DDG)
- place of the current loop in its loop hierarchy
- existing paths in the CFG of the loop
- unrolling information if detected

Currently CQA is fully integrated into the MAQAO framework and is available for use in the binary release [23]. Figure 3 summarizes the integration into MAQAO. In the next two sections we discover the performance model used by the tool and then the metrics on which the high level reports are based.

III. Performance modeling

This section explains how the performance model of an architecture is built. The Intel Sandy Bridge processor is taken as an example. The modeling of the execution pipeline is first detailed. Then the methodology giving the latencies and throughputs of instructions is exposed. After that the set of additional analyses used to refine the execution pipeline modeling are explained. The last part of this section proposes a mean to evaluate the real performance of a given loop in order to compare it with the CQA estimation which is actually a lower bound on execution time (optimistic).

A. Benchmarking latency and throughput of instructions

The MAQAO microbenchmark module, *microbench*, measures the performance of individual instructions provided by the MAQAO grammar generator and forwards it to the CQA module. In other words, for each element of an instruction sequence, CQA knows the corresponding impact on the selected machine model. Up to now most instructions are supported by *microbench*, in particular SSE/AVX instructions and basic general-purpose instructions involved in loop control and address calculation (ADD, SUB, CMP, conditional jumps). SSE/AVX are x86 vector instruction sets and must be used to harvest full performance from recent processors. For non supported instructions, Agner Fog's instruction tables are used.

The instruction list provided by the grammar generator is a text file, showing one line for each instruction. Instructions are described by their name (opcode) and operands (type and order). For each instruction, the module measures throughput and latency, that is cycles per instruction, by generating, compiling and running an assembly kernel structured as a loop. The body of this loop is a sequence of independent (for throughput) or dependent (for latency) instructions formed by the same base instruction (see figure 4(a) and figure 4(b)). In terms of throughput, the minimum number of independent streams to issue is the product of the latency by the number of execution units that can work in parallel. For example, on the Intel Haswell processor, two execution units can execute products with a 5 cycles latency, requiring at least 10 independent instructions streams to harvest full power (to fill all pipelines stages for both units). For latency, a RAW dependency must exist between each consecutive stream (the register written by a stream must be read by the next one).

Some instructions are more complex to process than others like DIV/SQRT since their latency depends on operands value. In that case:

- two loops must be generated for each measurement: one for best case (fastest) and another for worst case (slowest)
- these loops use dedicated operand values (typically 1.0 for best case and empirically found for worst case)
- to measure latency, the value of the destination operand must be reset at each iteration without breaking inter-iteration dependencies. It can be done by a MIN instruction, returning the minimum value of two operands

For out of order processors like Intel processors from Core 2 to Haswell, the module also measures for each instruction the number of fused (in the front-end) micro-operations (uops) and the dispatch of unfused uops (in the back-end) in execution ports. Hardware counters are used for that purpose.

For stable and precise measurements:

- the kernel is run before measurements to warm up buffers, caches and prefetchers
- the most precise and least intrusive timing method is used: RDTSC-based. RDTSC is an x86 instruction returning the number of reference cycles (cycles at reference frequency) elapsed since last reset
- the kernel is repeated multiple times during measurements to mitigate potential power saving mechanisms (after a long inactivity period, the hardware could require a long time, e.g. milliseconds, to recover full speed)
- the binary driving the kernel is repeated multiple (typically 31) times to report statistically valid measures (a median is significant only over a sufficiently big population)
- the binary is pinned on a core to get rid of migration overhead

```
L1:
  ADDSS %XMM0,%XMM0
  ADDSS %XMM1,%XMM1
  (...)
  ADDSS %XMM6,%XMM6
  ADDSS %XMM7,%XMM7
  SUB $8,%EBX
  CMP %EBX,%ECX
  JG L1
```
(a) Independent streams
. (throughput)

```
L2:
  ADDSS %XMM0,%XMM1
  ADDSS %XMM1,%XMM2
  (...)
  ADDSS %XMM6,%XMM7
  ADDSS %XMM7,%XMM0
  SUB $8,%EBX
  CMP %EBX,%ECX
  JG L1
```
(b) Dependent streams
(latency)

Fig. 4. Independent and Dependent streams (assembly code)

B. Execution pipeline modeling

Figure 5 shows an overview of the execution pipeline of the Intel Sandy Bridge processor. It is composed of in-order and out-of-order parts. The core of the out-of-order engine (i.e. the reservation station) is not modeled because of its complexity and lack of documentation. The other reason is CQA targets best performance (lower bound on cycles).

Each stage is modeled separately in order to have a modular and upgradable model. Thanks to this modular approach CQA supports all micro-architectures since Core 2 duo until the latest one, Haswell. To obtain an estimation of the number of cycles for a loop iteration, corresponding instructions are scanned and travel through the stages. Depending upon the state of the pipeline, bubbles can be inserted when suffering from a bottleneck. Additional penalties in term of cycles may be added in some specific cases.

Beyond the estimation of the number of cycles spent in one iteration of a given loop, CQA also gathers statistics which provide an insight within the bottlenecks. For example it is possible to see that a loop is stressing too many specific resources, which in most cases, is translated into a slowdown. CQA provides both front-end and back-end statistics.

CQA supports Xeon Phi via a simplified machine model due to the in-order flavor of this processor. Instructions can be decoded in two pipes U and V (with pairability rules) and vector instructions are executed by a single Vector Processing Unit (VPU) accessible via the U-pipe.

C. Static analysis

Thanks to advanced static analysis, the CQA performance model can be refined in order to better reflect reality.

1) Data Dependency Graph: Since our model does not simulate the reservation station stage of the execution pipeline, penalties due to data hazards must be taken into account. To address this issue, the data dependency graph of a given innermost loop is looked for true data dependencies (Read After Write) on register operands within two consecutive loop iterations. CQA can then estimate cycles taking them into account by considering the maximum cycles count between front/back-end and the critical path latency (longest dependency cycle). Consequently, the impact of loop-carried (inter-iterations) dependencies will be reported for affected loops (typically reductions into a scalar).

2) Loop hierarchy: The Intel Xeon Phi instruction set introduced scatter/gather operations on vectors. Always coupled with a mask-conditional branch instruction which generates a small loop, they mislead the innermost loop focus strategy of CQA. To avoid such pitfalls, these loops are discarded and only considered as scatter/gather instructions.

3) Paths: Depending upon the complexity of the control flow structure, innermost loops may have multiple paths from the entry to the exits. This is usually a clear sign of complex loop structure. Most of the time it is caused by if-statements. When experiencing such cases CQA provides an analysis for each path with a arbitrary limit of eight paths (this value can be changed).

4) Unroll factor heuristic based on arithmetic patterns: Reporting loop unroll factor (as applied by the compiler) is useful since loop unrolling can badly interact with source-level optimizations. For instance, one does not expect that the compiler unrolls a loop that was already unrolled at source level. For some pathological cases/patterns (typically spill/fill), CQA can advise to lower the unroll factor.

MAQAO detects loop unroll factor by comparing the arithmetic (or memory) footprint of the biggest (main) and the smallest (peel/tail) binary loop within the same source loop. If a source loop is composed of only one binary loop (non unrolled or unrolled with no peel/tail code) or binary loops with a similar footprint (multi-versioning), no unroll factor is provided.

The heuristic is specialized for Xeon Phi since peel/tail loops can be vectorized (by write-masking) or generated as scalar GATHER/SCATTER based loops.

5) Source and binary loops relationship: Source loops can be converted into potentially multiple binary loops. Figure 6 illustrates the correspondence between a source loop and the kind of binary loops that are usually found.

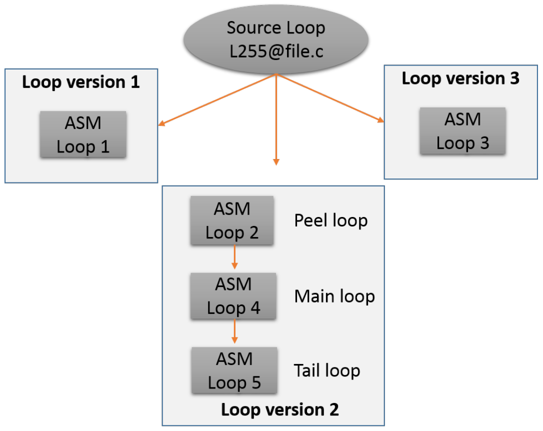

Fig. 6. Structure of assembly loops and corresponding source loop

This is usually due to the compiler choice of generating multiple strategies to be selected at runtime. It can also be split into multiple pieces, typically a peel loop, a main loop and a tail loop. In general peel and tail loops are used to treat cases not handled by the main loop (starting/ending iterations) when for instance the source loop was vectorized and/or unrolled.

D. Dynamic extension

In order to evaluate the gap between our prediction and the real number of cycles spent in a loop, an optional dynamic extension was added. It consists in a MIL [5] script and some post processing based on the returned data. The MIL script counts the number of cycles spent in a given loop. Then a post-processing Lua script compares the static estimation against the dynamic observed measure. The gap between these two metrics provides a valuable information on the degradation of the real performance since the static estimation considers data as resident in the first level cache.

IV. Architectural metrics and speedup opportunities

Code quality analysis at core level is only useful if speedup opportunities are identified for a given architecture. Indeed, it

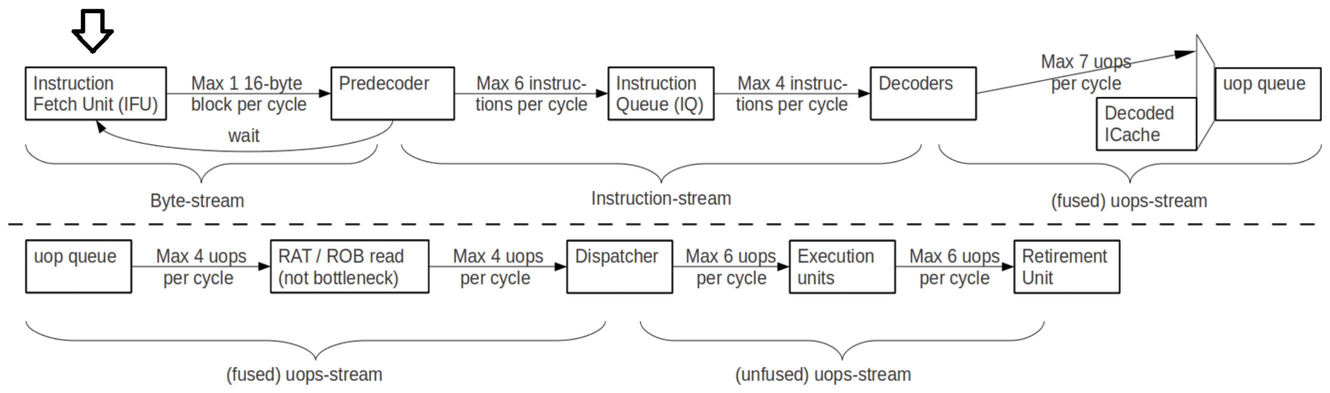

Fig. 5. Intel Sandy Bridge architecture execution pipeline model overview

is essential since metrics will evaluate the utilization efficiency of these opportunities.

A. Speedup opportunities

Speedup opportunities are defined as mechanisms by which it is possible to enhance the overall performance of a compute core. Sometimes these may imply some restrictions (e.g. arithmetical precision).

1) Vectorization: Vectorization is one of the most important levers when it comes to enhancing the performance of a code. Based on the instructions found in a given piece of code, a degree of vectorization is computed. Indeed, in order to exploit the vector facility, vector extensions must be used.

2) Low cost instructions: Some architectures provide fast instructions which can imply some restrictions. For instance if precision is not a major concern, recent Intel processors provide fast reciprocal instructions to lower the cost of the expensive DIV/SQRT instructions.

3) Data alignment: Most architectures have a better efficiency at reading data from memory when it is aligned on a given boundary. Using instructions which move non aligned data has an extra cost. Recent processors succeed in reducing the cost of such memory access but there is still a small difference that can become important if multiple instructions are concerned.

B. Metrics

1) Vectorization: The degree of vectorization will be computed as the ratio $\frac{vectorized\ instructions}{vectorizable\ instructions}$. It is even possible to produce multiple ratios by using a two-level categorization of vector instructions. The instruction set architecture (ISA) can manipulate single or double precision values in scalar or packed (vectorized) mode. A vector efficiency ratio is also provided in order to verify if the compiler has efficiently vectorized the code.

Another important fact to take into account is that, due to architectural limitation, mixing SSE and 256 bits AVX instructions induces a typical 75 cycles penalty and should be avoided. In that case, CQA displays the number of such transitions and the corresponding workaround.

2) Expensive instructions: Reporting expensive instructions provides users with a hint. The compiler may have generated a poor code pattern that the user is unaware of. The user can then investigate the issue. The following instructions are tracked:

- DIV/SQRT
- conversions
- unaligned memory accesses

3) Data dependencies: Presence of true data dependencies always has to be checked because it can drastically limit the performance. Sometimes it is possible to remove it.

4) Register pressure: Some code optimizations rely on the number of available registers. Determining the register pressure of a piece of code may provide interesting indications. For instance, if the register pressure is low then unrolling may be a good option. Alternatively, if the compiler resorts to spill/fill mechanisms and there are available registers, then it means that its register allocation heuristic failed to use all the available registers. The compiler actually uses an approximation and not the optimal solution since the problem is NP-Complete. One option is to switch to intrinsics if it is worth it. Note that this kind of optimization may really pay only if the L1 cache cannot host spill/fill values.

5) Compiler flags: The compiler remains the best tool to optimize an application. However it requires a lot of help. Some common flags are sometimes forgotten and the impact on the performance is catastrophic. For instance GNU and Intel compilers need a special flag in order to generate code for a specific architecture. That means that, even when using the most recent processors, the compiler does not generate the code that will take advantage of the available instructions, if that flag is not present. CQA handles the parsing of the compiler flags used to compile the given binary (when the information is available) and creates a report if an important flag is missing. Conversely some flags may provoke issues and this also can be notified.

C. Reports

Estimated cycles, execution bottlenecks and low level metrics would not be of much assistance to a user if presented as is. CQA provides human readable reports which are related to their source code when debug information is available in the binary. Detected issues are classified by confidence level. There are four of them:

- *high*. Provides information that most of the time leads to a performance gain.
- *potential*. The proposed optimization may be interesting.
- *hint*. No real match but provides the most important metrics that are outside the normal ranges.
- *expert*. Low level details only suitable for experts. For instance it provides the assembly code of a given loop and a breakdown of cycles spent in front-end (instruction fetch, decode...) and back-end (execution ports, DIV/SQRT unit and critical path latency).

CQA can also display the speedup that can be gained by solving some listed issues (vectorization, first bottleneck and slowdown caused by scalar integer instructions) under optimistic conditions (mainly: data in L1).

More information about the majority of advice provided by CQA is available in the tutorial section of MAQAO website [23]. Section VI presents examples of reports and how they are used.

V. Implementation and design decisions

In the two previous sections we have discovered the performance model used by the tool and also the metrics on which the high level reports are based. In this section both the design decisions that were taken along with the associated accuracy of the produced reports and the limitations of the performance model will be discussed.

A. Design decisions and limitations

The first design decision was to establish the target audience of the tool because presenting results on the analysis of assembly code was not obvious. The best response was to consider reports based on confidence levels, which naturally classify reports understandable by regular developers and of interest for experts. As stated at the beginning of this paper, the model is based on the hypothesis that data are resident in the first level cache and does not consider interactions between threads. Obviously the predicted performance gains will not be correct when data are resident in higher level caches or memory. The tool remains useful even if performance predictions are not correct. Detecting vectorization issues for instance remains very important because it generally requires a good memory layout. So it is important to take this information into account when working on memory issues (e.g.: reshaping data structures). In practice the tool can be used in conjunction with dynamic characterization tools (like DECAN [14]) that can measure to which extend computation issues matters.

Since our analyses are statically done there is no silver bullet to deal with branches typically caused by $if-statements$. The selected choice was to generate results considering each control flow path separately. It may sound simplistic but $if-statements$ are most likely to be a performance killer in a loop. One of the main goals of the tool is to provide an idea about the quality of the code. In this case that means that, either the compiler generated a poor quality code or the algorithm has to be changed.

Some issues are compiler specific and the tool only supports GNU and Intel compilers at the moment, which already represents a pretty high percentage of users.

B. Implementation difficulties

The tool is architecture dependent since it helps getting the best out of a given architecture and even micro-architecture. A special effort has been necessary to be able to properly handle architectures (and micro-architecture) dependent and non-dependent parts of the software.

Reports based on bad pattern detection are actually founded on the expertise of the developers. This supposes performing performance evaluation and optimization of real scientific applications using the different compilers and first finding issues *by hand*. The next step consists in automating the recognition of these issues in the form of high level reports. Many adjustments and testing have been necessary to reduce to a very low percentage false positives. For instance we use a data dependency graph to take into account penalties due to data hazards and inter-iteration dependencies.

VI. Case studies

CQA has been successfully used [12] as a basis for retrieving low level quality metrics. This section presents three case studies showing how to optimize applications thanks to CQA. The first two examples deal with real life scientific applications and have been directly used by the application developers. The last example demonstrates via CQA how an *a priori* simple code can perform poorly. These codes have been compiled with Intel compilers. Presented CQA reports are not exhaustive: only sentences that are useful to the current case study are selected.

A. YALES2

YALES2 [17] is developed at CORIA near Rouen, France. It is a numerical solver dedicated to the simulation of turbulent reactive flows with the Large Eddy Simulation method. It is a finite volume code which can deal with unstructured meshes and has an innovative 4th order spatial scheme for the discretization of convective and diffusive terms. It is based on the low-Mach number approximations of the Navier-Stokes equations and thus has to solve an elliptic Poisson equation at each iteration. A major feature of the code is that it implements a very efficient Deflated Preconditioned Conjugate Gradient (DPCG) to solve this problem [15]. It has been demonstrated to scale very well up to more than 16'000 cores. Indeed, the pure MPI version is based on a subdomain decomposition with adjustable domain size allowing a very efficient cache usage. Intel Fortran compiler 14.0.1 was used to compile YALES2, using *-O2 -xHost* flags (*-O3* provides no significant speedup). If no contrary precision, all results are for and on a Sandy Bridge machine. Analyzing the hottest loops with CQA reveals (typical example given figure 7) that most of them are not vectorized and the quality of their binary code is very low (lot of integer instructions, science being on floating-point values, and saturated load ports). Figure 8

```
Your loop is processing FP elements but is NOT OR
PARTIALLY VECTORIZED and could benefit from full
vectorization. (...) By fully vectorizing your
loop, you can lower the cost of an iteration from
7.00 to 4.38 cycles (1.60x speedup).
Two propositions:
 - Try another compiler or update/tune your
current one:
  * Intel: 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.

Detected a slowdown caused by scalar integer ins-
tructions (typically used for address computation).
By removing them, you can lower the cost of an it-
eration from 7.00 to 5.00 cycles (1.40x speedup).
To reference allocatable arrays, use
``allocatable'' instead of ``pointer'' pointers.
For structures, limit to one indirection. For
example, use a_b%c instead of a%b%c with a_b set
to a%b before this loop.
```

Fig. 7. Initial CQA report for the YALES2 typical loop

```
! for each element
do i = 1, size_outer

   elt => elts(i)

   ! for each pair
   do j = 1, size_inner

      ! integer, pointer :: val(:)
      ind1 = elt%p1%val(j)  ! node #1
      ind2 = elt%p2%val(j)  ! node #2

      coeff = a(j) * (b(ind2) - b(ind1))
      c(ind1) = c(ind1) + coeff
      c(ind2) = c(ind2) - coeff

   end do

end do
```

Fig. 8. YALES2 original loop

presents a typical loop with both indirect access and double structure indirections. Vectorization is limited/prevented by indirect access (in general not efficient without hardware support) and sub-optimal code quality is present each time source code contains double structure indirections or pointers targeting allocatable arrays. Indirect accesses in data structures cannot be avoided: they simply reflect the unstructured aspect of the discretization mesh [16].

To optimize YALES2, the typical loop presented in figure 8 is analyzed with CQA in different versions:

- original (with double structure indirections and generic pointers to access arrays)
- single_indir: obtained from original by replacing double with single structure indirections as shown figure 9.
- allocatable: obtained from single_indir by replacing generic pointers with allocatable variables

If a performance gain is predicted by CQA, YALES2 is recompiled and rerun after optimizing the hottest loop matching the typical loop pattern. If speedup is confirmed, the optimization is applied for other hot loops with the same

```
do i = 1, size_outer

   ! (...)
   elt_p1 => elt%p1
   elt_p2 => elt%p2

   do j = 1, size_inner

      ind1 = elt_p1%val(j)
      ind2 = elt_p2%val(j)
      ! (...)

   end do

end do
```

Fig. 9. YALES2 loop after replacing double indirections

metric	original	single_indir	allocatable
cycles	9	7	4.5
used x86 registers	15	16	9
stack references	0	4	0
nb instructions	32	22	13
bytes loaded	92	60	28
FP ops per cycle	0.44	0.57	0.89

TABLE I
MOST IMPACTED CQA METRICS DURING KERNEL OPTIMIZATION

pattern. One can see in table I than CQA estimates a 2x speedup from the original to the fully optimized version and the corresponding assembly codes in figure 10.

By optimizing both double indirections and allocatable pointers, the speedups described table II were obtained on three datasets warming completely different portions of the code.

Another way to optimize YALES2 is to address the vectorization issue mentioned in figure 7. Since it is difficult to change global data structures, CQA was used to statically estimate the potential speedup for a given transformation enabling vectorization. Forcing vectorization of the original loop via a pragma alters its semantic. This is caused by indirect accesses present in reductions. One approach consists in moving these reductions in two extra loops as presented in figure 11 by adding some arrays allowing a reverse access (getting pairs from nodes). The first loop contains indirections from pairs to nodes and precomputes **coeff** for all pairs. The second and third loops add to each node, **coeff** contributions coming from all matching pairs. In other words, computing **coeff** (from pairs and nodes) and affecting it to nodes is then decoupled, making vectorization possible. Vectorization has been forced by using SIMD directives on top of all innermost loops. The original loop costs 4.5 cycles per source iteration, according to CQA. The first loop of the transformed set of loops can not be vectorized, due to the indirection, and still

Appl. exe. time (s)	3D_cylinder	MS_1D_flame	1D_flame
original	111	56.6	84.9
optimized	63.7	36.85	68.9
speedup	1.74	1.54	1.23

TABLE II
APPLICATION SPEEDUPS OBTAINED ON THREE YALES2 DATASETS CORRESPONDING TO DIFFERENT PHYSICS

```
MOV      0x8(%R15,%RCX,1),%RSI
MOV      0x40(%RSI),%R13
NEG      %R13
ADD      %R11,%R13
IMUL     0x38(%RSI),%R13
MOV      (%RSI),%RDI
MOV      0x10(%R15,%RCX,1),%RSI
MOVSXD   (%RDI,%R13,1),%RDX
MOV      0x40(%RSI),%R13
NEG      %R13
ADD      %R11,%R13
INC      %R11
IMUL     0x38(%RSI),%R13
MOV      (%RSI),%RDI
MOV      %RBP,%RSI
MOVSXD   (%RDI,%R13,1),%R13
MOV      %RBP,%RDI
IMUL     %R13,%RSI
IMUL     %RDX,%RDI
IMUL     %RAX,%RDX
IMUL     %RAX,%R13
VMOVSS   (%RSI,%R14,1),%XMM0
VSUBSS   (%RDI,%R14,1),%XMM0,%XMM1
VMULSS   (%R10,%R12,1),%XMM1,%XMM4
ADD      %R8,%R10 / INC %R8
VADDSS   (%RDX,%RBX,1),%XMM4,%XMM2
VMOVSS   %XMM2,(%RDX,%RBX,1)
VMOVSS   (%R13,%RBX,1),%XMM3
VSUBSS   %XMM4,%XMM3,%XMM5
VMOVSS   %XMM5,(%R13,%RBX,1)
CMP      %R9,%R1
JLE      ad <myroutine_+0x9d>
```

Fig. 10. Original vs optimized assembly code. Instructions in italic have been suppressed in the optimized code and other instructions are very similar in both versions

costs 3 cycles per iteration. The second and third internal loop still cost 1 cycle (if vectorized) or 3 cycles (if not vectorized) per iteration. Moreover, due to the small number of iteration count in the inner loop ($\approx 5-15$), the overhead of the external loop can be important. Even when maximum vectorization is achieved, there is no gain to hope from this approach. These conclusions have been validated in the real application: the second approach is actually 2 to 3 times more CPU consuming than the original one.

B. *QMC=Chem*

QMC=Chem [19], [20] is a Quantum Monte Carlo application for Chemistry developed at Paul Sabatier University in Toulouse, France. It has very good peta- and exascale properties (embarrassingly parallel).

It presents two main hotspots: a matrix inversion (using DGETRF and DGETRI BLAS calls) and a hand-made sparse matrix-vector multiply (SPMV). Focus will be put on the second one since it is the only one that the application developer can reasonably optimize.

The SPMV routine multiplies a matrix A with 5 matrices B1-B5. The A matrix is then reused for the five multiplications but the loop is distributed in 3 loops: the first loop processing B1 and B2, the second B3 and B4 and the last one B5 (see figure 12, were B1 and B2 are processed). Figure 13 shows that the loop is vectorized but uses only half vector width (128 bits SSE instructions run on 256-bits AVX machine) and that a 2x speedup could be gained.

Following CQA propositions, vector arrays have been aligned and recompiled with $-xHost$ (micro-architecture specialization). After this optimization (figure 14), CQA reveals

```
!DIR$ SIMD
do j = 1, size_inner

   ind1 = elt%p1%val(j)  ! node #1
   ind2 = elt%p2%val(j)  ! node #2

   coeff(j) = a(j) * (b(ind2) - b(ind1))

end do

! for each node
do j = 1, size_inner2
   ! first and last pairs connected to node #1
   beg = rev_elt_p1_index (j)
   end = rev_elt_p1_index (j+1)
   coeff_sum = 0.

   ! for each pair connected to node #1
!DIR$ SIMD
   do k = beg, end-1
      coeff_sum = coeff_sum + coeff &
         (rev_elt_p1 (k))
   end do

   c(j) = c(j) + coeff_sum
end do

! similar to previous loop but on node #2
do j = 1, size_inner2
   beg = rev_elt_p2_index (j)
   (...)
end do
```

Fig. 11. YALES2 vectorizable version of the original loop

```
do j=1,LDA

  C1(j) =C1(j) +(  A(j,k_vec(1))*d11 +
                   A(j,k_vec(2))*d21 +
                   A(j,k_vec(3))*d31 +
                   A(j,k_vec(4))*d41)

  C2(j) =C2(j) +(  A(j,k_vec(1))*d12 +
                   A(j,k_vec(2))*d22 +
                   A(j,k_vec(3))*d32 +
                   A(j,k_vec(4))*d42)

enddo
```

Fig. 12. QMC=Chem loop computing A.B1 and A.B2

```
Your loop is vectorized (all SSE/AVX instructions
are used in vector mode) but on 50% vector length.

Assuming all data fit into the L1 cache, each
iteration of the binary loop takes 8.00 cycles. At
this rate:
- 50% of peak computational performance is reached
(8.00 out of 16.00 FLOP per cycle (GFLOPS @ 1GHz))

Your loop is processing FP elements but is NOT OR
PARTIALLY VECTORIZED (...).
By fully vectorizing your loop, you can lower the
cost of an iteration from 8.00 to 4.00 cycles
(2.00x speedup).

Propositions:
 - Pass to your compiler a micro-architecture spe-
cialization option:
   * Intel: use axHost or xHost.
 - Use vector aligned instructions:
  1) align your arrays on 32 bytes boundaries,
  2) inform your compiler that your arrays are
vector aligned:
    * Intel: use the VECTOR ALIGNED directive.
 - Use the LOOP COUNT directive
```

Fig. 13. Initial CQA report for QMC=Chem

```
Your loop is fully vectorized (all SSE/AVX
instructions are used in vector mode and on
full vector length).

The binary loop is loading 320 bytes (80 single
precision FP elements).
The binary loop is storing 64 bytes (16 single
precision FP elements).

Assuming all data fit into the L1 cache, each
iteration of the binary loop takes 10.00 cycles.
At this rate:
- 80% of peak computational performance is reached
(12.80/16 FLOP /cycle(GFLOPS@1GHz))
- 100% of peak load performance is reached (32/32
bytes loaded /cycle(GB/s@1GHz))

Load units are a bottleneck. Try to reduce the
number of loads. For example, provide more
information to your compiler:
 - hardcode the bounds of the corresponding 'for'
loop

used ymm registers      : 10 (low level metric)
```

Fig. 14. CQA report for QMC=Chem after recompiling with -xHost and aligning vector arrays

that the loop is now fully vectorized (on full vector length) but the bottleneck is now load units. According to source code only 48 elements could be loaded (if correctly reused from registers) but 80 were effectively loaded. And the low level metrics showed that only 10 registers were used (out of 16 available on x86_64). CQA proposes to reduce the number of loads by hard coding the loop bounds. Applying this optimization (LDA transformed into a constant defined at compile time), CQA reported 14 registers used, 48 loaded elements (figure 15) and a 100% computational resources usage (peak performance). Measured speedup was about 10% locally (matrix multiply) and 5-6% globally (application).

```
Your loop is fully vectorized (...).

The binary loop is loading 192 bytes (48
single precision FP elements).

Assuming all data fit into the L1 cache, each
iteration of the binary loop takes 8.00 cycles.
At this rate:
 - 100% of peak computational performance is
reached (16/16 FLOP /cycle(GFLOPS@1GHz))

used ymm registers      : 14 (low level metric)
```

Fig. 15. CQA report for QMC=Chem after hardcoding loop bounds

C. Complex

The code presented in figure 16 illustrates how CQA can help to optimize a very simple code (expected to use full processing power). The loop divides two complex arrays into a third one. Using Intel compiler, with typical flags (*-g -O*2 *-xAVX*), CQA detects that slow x87 code was generated (figure 17) and advises to use an option called complex-limited-range. Following this direction, code generation is much more efficient (figure 18). Execution time is divided by approximately 100, probably due to the cost of frequent commutations between the x87 and the SSE/AVX FPUs.

```
  COMPLEX(8), INTENT(OUT) :: a(1:n)
  COMPLEX(8), INTENT(IN)  :: b(1:n)
  COMPLEX(8), INTENT(IN)  :: c(1:n)

  DO i=1,n
     a(i) = b(i) / c(i)
  ENDDO
```

Fig. 16. Complex divide code

```
48 x87 instructions are processing arithmetic or
math operations on FP elements in scalar mode
(one at a time).

Detected X87 INSTRUCTIONS.
x87 is the legacy x86 extension to process FP
values. This instruction set is much less ef-
-ficient than SSE or AVX. In particular, it
does not support vectorization (x87 units not
being vector units).
(...)
Intel: if complex divides are used, use
complex-limited-range that is included in
fp-model fast=2 (see manual for safe usage).
```

Fig. 17. CQA report for complex divide using x87 instructions

```
Your loop is vectorized (all SSE/AVX instructions
are used in vector mode) but on 93% vector length.

The divide/square root unit is a bottleneck.
```

Fig. 18. CQA report for complex divide using AVX instructions

VII. Related Work

CQA is unique when considering quality analysis of code generated by compilers.

Intel IACA [11] appears to be the closest tool to CQA. It is quite similar in terms of performance modeling but differs in many ways. Like CQA, IACA computes static (optimistic) performance metrics for an innermost loop. In other words, these values can only be reached for an infinite loop with all of its operands in registers or in the first level cache. While CQA only needs a binary file, IACA requires recompiling source code in order to add special markers in the application. As far as we know, IACA does not simulate finely the front-end because it does not consider it as a possible bottleneck. CQA simulates each stage of the legacy front-end pipeline along with the micro-operations cache and the loop buffer. Furthermore, CQA provides the metrics output by IACA but it also provides a lot of extra metrics: vectorization ratio, average length of vectors used, arithmetic intensity, etc. CQA provides high-level reports aiming at helping the application developer (even with no knowledge on computer architecture and compilation) to transform its code or use more efficient compiler flags. Finally instead of displaying metrics at the instruction level, CQA displays metrics at loop level.

With respect to the latency and throughput information of instructions, other sources [9] [10] including IACA have their own methodology to obtain these.

Other tools using hardware performance counters or dynamic analyses can provides a restricted subset of the pathologies we can detect. VTune provides a few hints on some architectural bottlenecks based on hardware performance counters.

For instance replacing divisions by the product of inverse when the divide unit is saturated. Some tools [2], [8] based on dynamic analysis can detect vectorization oppotunities. The main difference of these approaches, when compared with CQA, is the need to execute the application and also a lesser number of pathologies detected.

VIII. Conclusion

This paper introduced the CQA tool, a loop-centric code quality analyzer tool based on architecture performance modeling and quality metrics. Performance modeling of the execution pipeline provides an estimation of the number of cycles spent in each iteration of a given innermost loop along with the architectural bottlenecks. The tool also analyzes the quality of the code generated by the compiler and provides high level metrics along with human understandable reports that relate to source code. Our modeling and analyses are statically done and do not require any application execution or recompilation. We show practical examples of situations where our tool is able to provide very valuable information leading to a performance gain.

For future work, an extension that covers the new ATOM (Silvermont) processors is planned. Another important target would be ARM. As mentioned before, CQA reports cycle estimations given the fact that data is in the first level cache. A nice extension would be to have estimations when data is in higher level caches up into the main memory.

IX. Acknowledgements

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 through 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] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs, 2008.

[2] O. Aumage, D. Barthou, C. Haine, and T. Meunier. Detecting SIMDization Opportunities through Static/Dynamic Dependence Analysis. Sept. 2013.

[3] S. Benedict, V. Petkov, and M. Gerndt. Periscope: An online-based distributed performance analysis tool. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Parallel Tools Workshop*, pages 1–16. Springer, 2009.

[4] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[5] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. S. Shende, A. D. Malony, and W. Jalby. MIL: a language to build program analysis tools through static binary instrumentation. In *20th Annual International Conference on High Performance Computing (HiPC'13)*, Hyderabad, India, Dec. 2013.

[6] I. Corporation. Intel®Advisor XE 2013, 2013. http://software.intel.com/en-us/intel-advisor-xe.

[7] I. Corporation. Intel®VTune^{TM}Amplifier XE 2013, 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[8] G. C. Evans, S. Abraham, B. Kuhn, and D. A. Padua. Vector seeker: a tool for finding vector potential. Sept. 2013.

[9] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. http://www.agner.org/optimize/instruction_tables.pdf.

[10] instlatx64.atw.hu. x86, x64 instruction latency, memory latency and cpuid dumps. http://instlatx64.atw.hu/.

[11] Intel. Architecture code analyzer. http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

[12] Y. Kashnikov, P. de Oliveira Castro, E. Oseret, and W. Jalby. Evaluating Architecture and Compiler Design through Static Loop Analysis. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE Computer Society, 2013.

[13] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis toolset. In M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Parallel Tools Workshop*, pages 139–155. Springer, 2008.

[14] S. Koliai, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby. Quantifying performance bottleneck cost through differential analysis. In *ICS*, pages 263–272, 2013.

[15] M. Malandain, N. Maheu, and V. Moureau. Optimization of the deflated conjugate gradient algorithm for the solving of elliptic equations on massively parallel machines. *Journal of Computational Physics*, 238(0):32 – 47, 2013.

[16] V. Moureau, P. Domingo, and L. Vervisch. Design of a massively parallel cfd code for complex geometries. *Comptes Rendus Mécanique*, 339(2-3):141–148, 2011.

[17] V. Moureau, P. Domingo, and L. Vervisch. From large-eddy simulation to direct numerical simulation of a lean premixed swirl flame: Filtered laminar flame-pdf modeling. *Combustion and Flame*, 158(7):1340 – 1357, 2011.

[18] V. Pillet, J. Labarta, T. Cortes, S. Girona, and D. D. D. Computadors. Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995.

[19] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby. Qmc=chem: A quantum monte carlo program for large-scale simulations in chemistry at the petascale level and beyond. In *VECPAR*, pages 118–127, 2012.

[20] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby. Quantum monte carlo for large chemical systems: Implementing efficient strategies for petascale platforms and beyond. *Journal of Computational Chemistry*, 34(11):938–951, 2013.

[21] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Openspeedshop: An open source infrastructure for parallel performance analysis. *Sci. Program.*, 16(2-3):105–121, Apr. 2008.

[22] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[23] M. Team. Maqao tool. http://www.maqao.org.

[24] B. J. N. Wylie and W. Frings. Scalasca support for mpi+openmp parallel applications on large-scale hpc systems based on intel xeon phi. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 37:1–37:8, New York, NY, USA, 2013. ACM.