## Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0</td>
<td>Initial version</td>
</tr>
<tr>
<td>v1.1</td>
<td>Several corrections, in particular typos and graph figures</td>
</tr>
<tr>
<td>v1.2</td>
<td>Better phrasing and graph caption corrections</td>
</tr>
</tbody>
</table>
D5.4 - MAQAO ARM Port
Version 1.1

Contents

Executive Summary 4

1 Introduction 5
   1.1 MAQAO for Performance Tuning 5
   1.2 Porting MAQAO to ARM 5

2 Preliminary Performance Study on ARM big.LITTLE 6
   2.1 Architectural context 6
   2.2 Impact of vectorization on energy on ARM32 8
      2.2.1 TSVC Benchmark 8
      2.2.2 Vectorization/Energy tradeoffs 9

3 Porting MAQAO on ARM 11
   3.1 Installation and usage 11
   3.2 Instrumentation 13

4 Performance Analysis with MAQAO on ARM 14
   4.1 Dependence Graph Building 14
      4.1.1 Register Dependence Analysis 14
      4.1.2 Detecting loop counters and induction variables 15
   4.2 Interpretation and Analysis 15
      4.2.1 Detection of Vectorization Opportunities 16
      4.2.2 Identifying Transformations 18

5 Performance Analysis 22
   5.1 TSVC benchmark 22
      5.1.1 Reductions 22
      5.1.2 Large Memory Strides 22
      5.1.3 Complex Control Flow 23
      5.1.4 Non vectorizable loops 23
   5.2 Porting and Optimizing SMMP with MAQAO 24
      5.2.1 Porting and profiling 25
      5.2.2 MAQAO Analysis 25
      5.2.3 Optimizing for SIMDization 26
   5.3 Single Precision Limitations for Other Applications 28

6 On-going Developments 29

7 Conclusion 32
Executive Summary

This deliverable presents the main features of MAQAO on ARM. After a study on the impact of vectorization and vectorization/energy tradeoffs on ARM32 architectures, we present the static analyses used on ARM and briefly the currently working instrumentation feature. Then we apply MAQAO on a benchmark in order to describe the hints given by the tool and apply MAQAO on SMMP, an MontBlanc application, in order to optimize it. Finally, we provide the on-going work concerning data layout transformations.
1 Introduction

This deliverable presents MAQAO [2] ported for ARM. We first describe a performance study on ARM, in order to capture the importance of some metrics on performance, then present MAQAO features implemented on ARM and finally, describe some performance study on MontBlanc applications with MAQAO.

1.1 MAQAO for Performance Tuning

MAQAO is a performance tuning tool for multicore codes. It analyzes binary executable codes and provides performance reports and hints to the users. These reports are obtained through the combined use of global static analyses on the binary and dynamic information captured through binary instrumentation.

For the static analyses, MAQAO finds functions, loops, control flow in the code, detects SIMDization and possible other optimizations performed by the compiler (inlining for instance). MAQAO proposes a C API to disassemble and analyze binary codes. All analyses can be extended and scripted with LUA. The SIMD analyses for instance proposed in this deliverable are written in LUA.

To capture the dynamic behavior of a code, MAQAO proposes an instrumentation API, able to perform time and value profiling. The functions of this API can be given parameters coming from the global static analysis (such as the depth of a loop, whether the code is part of an inlined function, ...) and coming from registers. The probes are user defined and can be either defined in assembly, in dynamically loaded libraries or even in LUA (and compiled through a JIT). This API is described for Intel architecture in a previous paper [4]. Value profiling of data streams (load and store addresses) can be handled with the compression proposed by NLR [6]. The combination of static and dynamic results is performed statically.

MAQAO consists in different modules, and we discuss in the following section the modules that need to be ported.

1.2 Porting MAQAO to ARM

MAQAO has been developed for a large range of Intel architectures, in collaboration with Intel in particular. The objective of this deliverable is to port MAQAO to ARM architecture.

Figure 1 gives an overview of MAQAO. MAQAO consists in:

- A binary parser: This element is central for an ARM port. As there exists different ISAs for ARM processors, we will focus on Arm32 instruction set (not thumb), and later on Arm64.

- The code representation is built through 3 key analyses: (i) The call graph, finding which function are called within each function, (ii) The control flow graph, detecting the execution paths within a function and (iii) The dependence graph, defining the partial execution order between instructions, to preserve for most optimizations. This part is also dependent on the architecture, due to particularities in the ISA for the description of the returns for instance or for post-increments. Besides, the instrumentation, based on the information collected by these analysis, relies on the ISA and has to be ported.
The performance analysis module generates the hints provided to the user. For this port, a specific analysis is proposed. The first part, presented in this document, corresponds to the SIMD analysis. The second part, that will be part of the second deliverable, will focus on data layout analysis and optimization.

Therefore, the ARM port requires to rewrite many MAQAO modules. However, two parts are essential for the rest of the development on ARM: The binary parser and the instrumentation API. The performance analysis will be developed during the whole project.

2 Preliminary Performance Study on ARM big.LITTLE

The target architecture is ARM big.LITTLE processors. This is an heterogeneous architecture with different key metrics driven by frequency, cache capacity, SIMD instruction efficiency. We highlight in this section the key features of this architecture, in terms of performance and energy consumption. The test have been conducted on a ARM32 big.LITTLE architecture, and tests on ARM64 big.LITTLE are still in progress.

Instead of modeling one particular architecture of ARM processor, our objective is to capture essential metrics and measure performance (instead of predicting it). To this intent, this first preliminary study shows how SIMDization is related to energy.

2.1 Architectural context

The ARM big.LITTLE architectures have been designed in order to be able to deliver performance and to save energy, according to the need. For our experiments we have two architecture setups, one with ARM32 Cortex-A15/Cortex-A7 processor (Samsung ODROID-XU), one with the ARM64 Cortex-A57/Cortex-A53 processor (ARM Juno development board). Figure 2 shows the two processors Cortex-A57/Cortex-A53 (with possibly the GPU) sharing an interconnect able to maintain cache coherency between the 2 L2. The same architecture holds for the Cortex-A15/Cortex-A7 of the Samsung ODROID XU+E board we used in our benchmarks.
The decision to execute a code on the big or LITTLE processor can be taken automatically by the system or imposed by the developer. When governed automatically by the system, the execution is switched from big to LITTLE or the reverse according to some heuristics, based on the current load of the machine. On the ODROID XU+E board, only the active processor is on, the inactive one is off. Only one of the two Cortex works at a time. Here, changing from one processor to the other at some point of an application requires to flush out data from one of the L2 cache in order to transfer them to the other L2 cache. The different steps required to perform this transfer are explained in Figure 3, including task migration. On more recent configurations, it is possible to use both processors at the same time.

The board we used for the development of MAQAO on ARM has been so far the ODROID XU+E, since we received it in February 2014. We received the ARM64 Juno development card in January 2015 and have not yet achieved a performance study on it. The ODROID XU+E is a Samsung development board with 2GB of RAM, featuring the Exynos5Octa processor with a quadcore Cortex-A15 (from 800MHz to 1.6GHz, 2MB L2) and a quadcore Cortex-A7 (from 250MHz to 1.2GHz, 512KB L2). Both processors have out of order execution. The PowerVR
for (int i = 1; i < LEN; i++)
    X[i] = Y[i] + 1;

Figure 4: function s000, TSVC

for (int i = 0; i < LEN2; i++)
    for (int j = 0; j < i; j++)
        aa[i][j] = aa[j][i] + bb[i][j];

Figure 5: function s114, TSVC

GPU available on the board has not been used. Switching from one processor to the other requires to operate both at 800MHz. On this processor, SIMD vectors are 128bits (4 floats single precision) and the processor has no native double-precision instruction. The operating system is Linux Ubuntu 12.04 LTS Linaro and the compiler used is GCC 4.6.3.

2.2 Impact of vectorization on energy on ARM32

Vectorization is one of the key optimization for high performance on modern architectures. We evaluate here the performance of both processors for vectorization.

2.2.1 TSVC Benchmark

The Test Suite for Vectorizing Compilers (TSVC) [3] is a benchmark suite of small kernels exercising the capacity of compilers to vectorize. It has been improved in 2011 [7] and rewritten in C. It has 151 kernels, mostly simple loops, and most of them can be vectorized. Figures 4, 5 and 6 show sample kernels from TSVC. Figure 4 represents a straightforward example of vectorizable code, 5 exhibits a triangular iteration domain and 6 has a non-trivial control flow graph, with an induction variable (j). All of them are vectorizable on modern architectures, but require different techniques so that SIMDization is efficient. The dataset is small enough to fit into the L2 cache.

We first try to evaluate the impact of vectorization on performance and energy, and thus consider kernels for which GCC is able to generate both versions. We have selected 10 simple kernels from TSVC that the compiler GCC is able to vectorize (checked thanks to the vectorization reports): va, vpe, vtv, vptv, vpets, vpepv, vtev, usumr, vdotr, vbor. The expression

int j = -1;
for (int i = 0; i < LEN; i++)
    if (b[i] > (float)0.)
        j++;
    a[j] = b[i] + d[i] * e[i];
else
    j++;
    a[j] = c[i] + d[i] * e[i];

Figure 6: function s124, TSVC
computed by each kernel is shown in Table 7 with its number of floating point operations. We

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Expression</th>
<th>Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>va</td>
<td>a[i]=b[i]</td>
<td>0</td>
</tr>
<tr>
<td>vpv</td>
<td>a[i]+=b[i]</td>
<td>$64 \times 10^9$</td>
</tr>
<tr>
<td>vtv</td>
<td>a[i]*=b[i]</td>
<td>$64 \times 10^9$</td>
</tr>
<tr>
<td>vpvtv</td>
<td>a[i]+=b[i]*c[i]</td>
<td>$25.6 \times 10^9$</td>
</tr>
<tr>
<td>vpvts</td>
<td>a[i]+=b[i]*s</td>
<td>$6.4 \times 10^9$</td>
</tr>
<tr>
<td>vpvvpv</td>
<td>a[i]+=b[i]+c[i]</td>
<td>$51.2 \times 10^9$</td>
</tr>
<tr>
<td>vttvtv</td>
<td>a[i]=b[i]*c[i]</td>
<td>$51.2 \times 10^9$</td>
</tr>
<tr>
<td>vsumr</td>
<td>sum+=a[i]</td>
<td>$64 \times 10^9$</td>
</tr>
<tr>
<td>vdotr</td>
<td>dot+=a[i]+b[i]</td>
<td>$64 \times 10^9$</td>
</tr>
<tr>
<td>vbor</td>
<td>all combinations</td>
<td>$12,288 \times 10^9$</td>
</tr>
</tbody>
</table>

Figure 7: Expressions and Flops for simple TSVC kernels

explore two dimensions for these kernels: their performance on the little and BIG processor, and the energetic consumption. In particular, we want to determine the gain in terms of energy provided by the vectorization and if trade-offs are interesting to explore in the context of HPC applications.

2.2.2 Vectorization/Energy tradeoffs

Two metrics have been considered: instantaneous consumption, showing the cost in terms of energy of the vector/scalar pipelines. And the global consumption, taking into account the time to execute a kernel. Figure 8 show results of the instantaneous consumption, both on ARM A15 and ARM A7. Figure 8 shows that the vector pipeline of the Cortex-A15 requires more energy than the scalar one, with a mean measured gap of 0.5 Watt. This gap for the Cortex-A7 processor is too small to be accounted for.

The global consumption determines whether vectorizing pays off, in terms of energy but also in performance measured in Gflops/cycle. Figure 9 shows the impact on energy for the A15 and A7. This is confirmed by the plots shown in the previous figure 9, vectorization has an important impact on performance and on energy, even if the vector pipeline requires more energy.
The timing measurements obtained (not shown here) follows the same pattern. Kernels consuming less are kernels that execute faster. Between the two architectures, the energy requirement is within a mean factor of 1.8 (A7 consumes less than the A15). However, some kernels such as vpvtv, vpvpv, vtvtv have similar energy consumption for A7 and A15, while they are faster on A15. One of the possible reasons is that the A15 architecture may be more efficient with three operand kernels.

The previous plots of the A7 and A15 in Figure 9 enabled the comparison between two versions: vectorized and non vectorized. Now, Figure 10 compares the efficiency of the same version of the kernels (vectorized), on both architectures. Figure 10 on the left shows the Gflop/W ratio and the right figure presents the Gflop/W/s metric. The first figure therefore focuses on the energetic efficiency of the code on the architecture. Globally for these simple kernels, the A7 outperforms the A15 in terms of energy efficiency (for vsumr, vbor kernels) but for the kernels cited previously (vpvtv, vpvpv, vtvtv), both architectures have the same consumption behavior.

In terms of efficiency measured in Gflop/W/s the Cortex-A15 outperforms nearly all the time the Cortex-A7, but there are still examples where both architectures are on par. Determining precisely the conditions on the kernels that lead to such a situation is still to be done. Figure 11 provides the speed-up on execution time, comparing both architectures for the same (vectorized) kernels. Combining these observations with the previous ones, Figure 11 shows that the Cortex-A15 offers a speedup of 2.2 compared to the Cortex-A7, even if for the vsunr kernel, both architectures provide the same performance in terms of GFlop/W/s. For most of the other kernels, the Cortex-A15 offers better Gflop/W/s and better performance.

This study shows that SIMDization is indeed critical for performance, on both big and
LITTLE processors. The SIMD pipeline of the benchmarked architecture consumes more energy (around 10%) than the scalar one but in terms of energy efficiency, even for simple kernels, performance between big and LITTLE architectures is in general in favor of the big processor due to the reduced execution time. In the following, only the big architecture will be considered. SIMDization is the main metric used by the current version of MAQAO on ARM.

3 Porting MAQAO on ARM

We provide thereafter a short description on how to use MAQAO on ARM32 architecture and what kind of result it provides.

3.1 Installation and usage

The prerequisites for MAQAO are provided in the README file:

- gcc and g++
- cmake with a version higher than 2.8.8

To build MAQAO inside the build directory, type:

```bash
> cmake .. -DARCHS=arm
> make
```

Other usual flags can be used for the cmake command. By default, MAQAO and its libraries are built inside the bin and lib directory. A system-wide installation can be obtained by using make install. The environment variable MAQAO_SA_PATH must be set to the source directory of MAQAO. MAQAO can be either compiled on ARM architecture or on an x86 Intel machine (faster).

MAQAO disassembler can be tested through the use of

```bash
> bin/maqao madras -d <mybinaryfile>
```
The binary files considered should be compiled with the -marm flag. Thumb instructions are not analyzed by MAQAO (even if they are disassembled).

To generate a report,

```
> bin/maqao simd_analyzer.lua <mybinaryfile>
```

with `mybinaryfile` the executable to analyze. The output generated corresponds to the analysis of the innermost loops of all functions. For a more pinpointed analysis, the name of the function to analyze can be given as parameter:

```
> bin/maqao simd_analyzer.lua runvec:s242
```

MAQAO then generates a report for the function named `s242` of binary called `runvec`. This report is:

```plaintext
analysing: s242
s242 debug data unavailable
```

```
---
- raw instruction listing -
  o l.137 0xf160: vldr s15, [r5, #0] ; flags(0x4010)
  o l.137 0xf164: ldr r5, [pc, #208] ; flags(0x10)
  o l.137 0xf168: ldr ip, [pc, #208] ; flags(0x10)
  o l.137 0xf16c: mov r3, #1 ; flags(0x10)
  o l.137 0xf170: ldr r0, [pc, #204] ; flags(0x10)
  o l.137 0xf174: ldr r1, [pc, #204] ; flags(0x10)
  o l.137 0xf178: mov r2, r5 ; flags(0x10)
  o l.137:136 0xf17c: vadd.f32 s15, s15, s16 ; flags(0x4010)
  o l.137:136 0xf180: add ip, ip, #4 ; flags(0x10)
  o l.137:136 0xf184: vldr s12, [ip, #0] ; flags(0x10)
  o l.137:136 0xf188: add r0, r0, #4 ; flags(0x10)
  o l.137:136 0xf18c: vldr s13, [r0, #0] ; flags(0x10)
  o l.137:136 0xf190: add r1, r1, #4 ; flags(0x10)
  o l.137:136 0xf194: vldr s14, [r1, #0] ; flags(0x10)
  o l.137:136 0xf198: add r3, r3, #1 ; flags(0x10)
  o l.137:136 0xf19c: cmp r3, #32000 ; flags(0x10)
  o l.137:136 0xf1a0: vadd.f32 s15, s15, s12 ; flags(0x10)
  o l.137:136 0xf1a4: vadd.f32 s15, s15, s13 ; flags(0x10)
  o l.137:136 0xf1a8: vadd.f32 s15, s15, s14 ; flags(0x10)
  o l.137:136 0xf1ac: vmov lr, s15 ; flags(0x10)
  o l.137:136 0xf1b0: str lr, [r2, #4]! ; flags(0x10)
  o l.137:136 0xf1b4: bne f17c ; flags(0x19)
  o l.137 0xf1b8: ldr r0, [pc, #124] ; flags(0x10)
  o l.137 0xf1bc: ldr r1, [pc, #124] ; flags(0x10)
  o l.137 0xf1c0: ldr r2, [pc, #124] ; flags(0x10)
  o l.137 0xf1c4: ldr r3, [pc, #124] ; flags(0x10)
  o l.137 0xf1c8: str r6, [sp, #16] ; flags(0x10)
  o l.137 0xf1cc: stm sp, r7, r8, s1 ; flags(0x10)
  o l.137 0xf1d0: str sb, [sp, #12] ; flags(0x10)
  o l.137 0xf1d4: blx 8814 ; flags(0x12)
  o l.137 0xf1d8: subs r4, r4, #1 ; flags(0x10)
  o l.137 0xf1dc: bne f160 ; flags(0x19)
```
circuits analysis - checking whether data dependences are compatible with vectorization
l.136 has 1 dependence circuit(s) on FP instructions
analysing dependence circuit 1 of l.136:
  0xf17c: vadd
  0xf1a0: vadd
  0xf1a4: vadd
  0xf1a8: vadd
> reduction with instruction vadd
> loop is vectorizable with reduction

generation of dot files -
----->./cfg_runvecs242.dot

The first part provides the listing of the assembly code analyzed. The left annotations, named l.137 and 136, show the scope of two nested loops. Here, loop 136 is included in loop 137. SIMD analysis is only performed on innermost loops. Then follows a list of analyses. Here on the previous example, the circuit analysis finds that there is a loop-carried dependence cycle involving floating point (FP) instructions. As all instructions involved are additions, this is a reduction and can therefore be vectorized. The list of analysis results are discussed in the following section. Finally, the name of the data dependence graph for the innermost loop is generated and its name is given at the end of the report. The file generated are dot files, and png figures can be obtained using the tool graphviz and the dot command:

> dot -Tpng ./cfg_runvecs242.dot

An example can be seen in Figure[12]

3.2 Instrumentation

MAQAO is able to patch ARM binaries. For this preliminary version, the patch can only be done for ARM32 architectures. The command line is the following:

> bin/maqao madras <mybinaryfile> --function="test;@0x8490"

where the argument --function defines the function probe to insert and the address where the call will be inserted. The function (here test) has to be defined in the binary code statically. This means that the instrumentation functions have to be given at compile time and the code need to be recompiled. Another approach, easier to use and with no need for recompilation, would require to dynamically load the probe functions from a library. The dynamic loading of an instrumentation library containing user probes is not yet functional.

The address given to the patcher can be automatically found from MAQAO according to user constraints. This low-level patching method can be used within MAQAO lua scripts, where the API can iterate through the code structure (loops, instructions, blocks) and provide the corresponding code addresses.

The patching mechanism proceeds by block relocation. The whole basic block, containing the address to patch, is relocated to a new code section. A jump is inserted at its previous address and the block is padded with nops. Registers are saved before calling the probe function and restored after its execution.
4 Performance Analysis with MAQAO on ARM

The main objective of this deliverable is to analyze ARM binaries and detect opportunities for optimization. MAQAO analyzes binary executable codes, restructures them by finding functions, loops, blocks and computes dependence graph. From this information, user hints are generated, finding possible ways to improve the code.

The construction of the control flow graph and call graph are based on usual techniques and is not described here. The disassembling itself relies on objdump library (libopcode) and is adapted for MAQAO. The dependence graph is detailed in the following, with the hints that are generated by MAQAO.

4.1 Dependence Graph Building

We propose a method to build statically the dependence graph between instructions. This method uses reduction detection with a register dependence analysis, performed statically on the code by MAQAO. Using instrumentation, a dynamic memory dependence graph will be used further in the project in order to better capture memory dependences.

4.1.1 Register Dependence Analysis

This dependence analysis is performed by MAQAO on instructions in inner-most loops. It computes existing dependences between any couple of instructions in the loop, due to the use of registers. The dependences are among one of the three types: RAW (read after write), WAR (write after read) and WAW (write after write). Here for the vectorization analysis, only the RAW dependence type is analyzed (true dependences) since WAW and WAR dependences are due to register reuse and can be removed by choosing a different register allocation. All register dependences and their distances (0: dependence inside the same iteration, 1: the write occurs one iteration before the read) are computed.

Several particularities of the assembly code are taken into account:

- Zeroing registers: when XOR-like instructions are applied with two operands that are the same register, this register is initialized to 0. The outcome of the instruction does not depend on initial input value of the register, even if there is a read, hence there is no dependence with previous instructions. This special case is used by compilers to initialize registers, in particular SIMD registers. Our dependence analysis takes this particular case into account.

- Post-incrementing address registers: on ARM, address registers can be post-incremented with LDMIA/STMIA instructions for instance, or more generally a ! suffix.

- Return and call instructions are usually achieved through explicit manipulation of the instruction pointer register. In particular, returns are usually generated through the pop instruction, restoring its value from the stack.

Besides, when registers are used by instructions for memory indices, the dependences to these instructions for these registers are tagged as memory index computation. This will be used later to separate the computation that need to be vectorized from the induction variable/address computation.

Figure 12 presents the code and dependence graph for function s000 from TSVC benchmark. The nodes in the graph are assembly instructions, the edges are dependences (RAW) with their distance.
```c
for (int i = 0; i < lll; i++)
    X[i] = Y[i] + 1;
```

Figure 12: Source code and dependence graph on the binary code of function s000. Labels on edges represent dependence distance. Red edges are dependences for registers used in address computation. Two recurrences occur, one with a post-incremented address register, the other with the loop counter.

### 4.1.2 Detecting loop counters and induction variables

Induction variables are detected through the analysis of the dependence graph, taking into account the specificities of some instructions (such as post-increments). Following the usual analyses proposed in the literature and implemented in compilers, dependence cycles are detected and whenever all instructions involved are used for address computation, if the instructions are only \textit{mov} and simple arithmetic operations the registers written are tagged as induction variables for the address computation. Capturing the stride of these variables is important to grasp how data structures are iterated.

This approach is simple but cannot capture all cases. To complement it, a trace-based approach using instrumentation is planned. This could capture more complex memory access patterns. Figure 13 shows an example of induction variable detection. Two independent registers are used to compute the 4 addresses (3 loads and one store), both use a 1024 increment, that advocate for either an optimization at the loop level (here an interchange with the outerloop) or a restructuring of the data layout (transpose).

### 4.2 Interpretation and Analysis

The dependence graph, associated to the knowledge of the individual instructions and register accesses provided by MAQAO, constitutes the foundation for the vectorization analysis. In this section, we describe the methods used to identify different vectorization opportunities on codes, in particular on TSVC benchmark suite [7]. Besides, we propose a set of hints in order to guide the user towards vectorization.
for (int j = 0; j < LEN2; j++)
    aa[j][i] = aa[j][i] + bb[j][i] * cc[j][i];

Figure 13: Source code and dependence graph on the binary code of function s2275. Two induction registers are detected, r2 and r3, with increment of 1024 in both cases. This corresponds to a large stride in the data structure.

4.2.1 Detection of Vectorization Opportunities

The method to detect possible vectorization is based on the dependence graph, with memory and register-based dependences and only determines whether a schedule exists for a possible vectorized version. The first step is to separate (if possible) in the graph the instructions relative to the computation of memory addresses or induction variables from the rest of the computation. The existence of a schedule for a vectorized version will depend on structural conditions on the remaining graph. We describe in details the different steps of this method.

Identifying Address Computation Vectorization concerns only a fraction of the instructions, some instructions are necessary for the control and for address computation and these will not be vectorized. This is therefore essential to separate in the dependence graph these different instructions. By tagging RAW dependences for registers used in address indexing (loads and stores), we partition the instructions belonging to the same connected component by cutting these edges. One of the partition corresponds to instructions relative to address computation while the second corresponds to the instructions to vectorize. Post-incremented address register are handled specifically: for such instructions (usually a load or store), two nodes are considered, one reading and writing the address register, the other performing the load/store.

This edge cut may lead to more than 2 partitions of the same connected component. This reflects the case where there are indirections (such as A[B[i]]). This property will be used in the following section to suggest data reshaping. Moreover, it is possible to find dependence graph where no cut exists. This does not occur in TSVC benchmark suite, but the following example illustrates this case:

for i = 0, n
    A[i] = B[i-1];
    B[i] = B[A[i]]
In this code, there is a dependence cycle between the two instructions, and one of the dependences is due to address computation. In this case, we cannot partition the dependence graph and the following conditions for vectorization will be applied on the whole graph.

![Dependence Graph](image)

Figure 14: Partitioning the dependence graph of function s171. The first partition corresponds to address computation, the second one to floating point computation. Red edges connect the two partitions.

Figure 14 represents the dependence graph on the binary code of function s171. The graph is partition so as to analyze separately the floating point part of computation from the address computation. Only the floating point instructions will be vectorized.

**Dependence Cycles** We only consider here partitions of the dependence graph that do not contribute to address computation (if partitioning is possible). We present in this section a sufficient condition for vectorization, based on the preservation of the dependences.

Dependences are weighted by their distance, and cycles in the dependence graph have cumulative weight > 0 (assuming single dimension distance vector). We assume the vectorization we want to achieve will place inside the same SIMD vector, the data accessed by a few consecutive iterations, such 4 floats for instance for Neon 128bit vectors. These 4 floats that were accessed through different iterations, are accessed after vectorization during the same iteration. In terms of dependence distances, the vectorization decreases the distance by a factor corresponding to the number of elements in a SIMD vector (4 in our example). Hence vectorization is possible if there exists a schedule for the vectorized dependence graph. Such schedule exists if and only if there is no cycle of weight 0. For multidimensional distance vectors, only the distance of the innermost loop is considered (the one that will be vectorized). Large strides or dependence cycles for this dimension may lead to consider loop interchange or transposition.

This condition is a structural property of the graph and can be automatically checked. In the current state of dependence graph computation, as only scalar dependences are computed (with 0 or 1 distance), this vectorization check is not necessary. However, as we plan to use memory traces to capture more memory dependences, this check will need to be performed.

**Reductions** We present in this section another sufficient condition for vectorization, this time not preserving dependences. Similarly to the previous section, we consider in the dependence graph the cycles with a weight lower than the size of the SIMD vectors (in number of elements). These cycles would be transformed after vectorization into 0-weight cycles, preventing from finding a schedule.
When all instructions involved in such cycle are the same associative operation (such as ADD, MAX, MIN) and the cycle is elementary, the computation boils down to a reduction: dependences can be broken and the computation can be rescheduled thanks to associativity. Depending on the dependence distance on the cycle, this will require some data layout transformation or shuffling (depending on SIMD ISA).

```
for (int i = 0; i < LEN; i += 5)
    dot = dot + a[i] * b[i]
    + a[i + 1] * b[i + 1]
    + a[i + 2] * b[i + 2]
    + a[i + 3] * b[i + 3]
    + a[i + 4] * b[i + 4];
```

Figure 15: Reduction detection: code and dependence graph for function s352. Address computation instructions have been removed. The cycle of length 4 is a reduction with a combination of vadd and vmla.

Figure 15 illustrates the case where the dependence graph has a cycle of weight 1, and all nodes in the cycles are additions (either only addition, or combined with a multiply). The code presented is not vectorized (using scalar registers instead of vectors). Therefore, this code computes a reduction and is vectorizable provided that this 4 term reduction can be rewritten with SIMD vector code. For this particular case, code is unrolled and each memory access has large strides. MAQAO will not find the reroll transformation but suggest to change the data structure (like changing array of structure into structure of array), and implement the reduction.

### 4.2.2 Identifying Transformations

In addition to the analysis of the dependence graph, MAQAO can help the user determining the transformations in order to make the code vectorizable.

**Instruction Scheduling** The dependence graph has no cycle and the code can be vectorized, provided that the instructions are scheduled in the loop according to the dependences. With intrinsics, load and store operations can be scheduled independently of computation operations. MAQAO can help the user finding a correct schedule for intrinsics instructions, in particular in presence of loop-carried dependences in the original code.
for (int i = 0; i < LEN-1; i++)
a[i] = b[i] * c[i] * d[i];
b[i] = a[i] * a[i+1] * d[i];

Figure 16: Instruction scheduling: code and dependence graph for function s241 (address computation have been removed for clarity). This function is vectorizable. All the loads involved in the multiplications have to be scheduled before the stores. However, the compiler has not succeeded the vectorization.

Figure 16 shows for function s241 a code that can be vectorized with no difficulty. There are two expressions computed here, sharing some common variables. The compiler has scheduled the left computation first, and then the second one. This prevents from vectorizing. The graph also shows a valid schedule where all loads are first scheduled (vectorized) and then perform all computations. MAQAO can here advise to reschedule explicitly the source instructions in order to ease vectorization.

**Data Reshaping** MAQAO static dependence analysis finds the strides used by the address registers. This provides several hints of transformations:

- The value of $s_n$ is larger than the size of an element accessed: either another stride $s_k$ has a value equal to the size of an element, in this case this advocates for a loop interchange, or a data layout transformation corresponding to a transposition. Or $s_n$ is the smallest stride, but it does not correspond to the size of an element (such as for function s2275, presented in Fig.13 where strides for all accesses are 1024-byte long). In this case, there are “holes” in the structure that need array reshaping. This generally corresponds to a change of array of structures into a structure of arrays.

- The value of $s_n$ is negative: negative strides may prevent the compiler from vectorizing. If all other memory accesses have also negative strides for the same loop, loop reversal can be a solution. For function s112 for instance, all loads and stores have strides of $-4$ (in bytes). Loop reversal is therefore possible here. For function s122, only one load has...
a negative stride and this advocates for changing the data layout of this array.

Loop Transformation  The dependence graph can show some opportunities for loop distribution, loop reversal (see data reshaping) or loop interchange. Loop distribution will be beneficial when one part of the computation is sequential while the other is parallel, and these parts can be separated. This occurs for function s222 illustrated in Figure 17. The cycle in the left one

```
for (int i = 1; i < LEN; i++)
    a[i] += b[i] * c[i];
    e[i] = e[i - 1] * e[i - 1];
    a[i] -= b[i] * c[i];
```

Figure 17: Loop distribution: code and dependence graph for function s222. The loop has two distinct slices of computation. It could be distributed into two loops.

is not a reduction but is only due to post-increment instruction. This part is vectorizable. On the right, the loop is sequential due to memory dependence (not detected here by the static dependence analysis).

Reduction Rewriting  Reduction detection should generate a hint showing how to write a reduction with intrinsics. The code will depend on the SIMD ISA used.

Idiom Recognition  Idiom recognition consists in recognizing a (vector) expression from the dependence graph. Using the memory trace to identify the different memory arrays used by the computation, and the register-based dependence graph to identify the vector operations, it is possible to detect the following operations from TSVC benchmark (X, Y denote vectors, c scalar) and build the vector expression, as a hint for the user:

- memcpy, on dense or sparse arrays
- reductions such as $c = X[i]$ (s311), $c = \text{dot}(X, Y)$ (313), $c = \max(c, \max(X))$ (314)
- vector operations such as $X = Y + c$ with $c$ a scalar (s000), $X[i] = Y[i] + X[i - 1]$ (111), and all functions $\text{vpv}, \text{vtv}, \ldots$

The hint for all the functions among which we can associate a library function, is to replace the function with the appropriate call.
Figure 18 shows a dependence graph where the heart of the computation is a mem copy, and the array read is accessed through an indirection. The user then has to determine the appropriate library call to replace this code, according to the indirection array.

```c
for (int i = 0; i < LEN; i++)
    a[i] = b[ip[i]];
```

Figure 18: Idiom recognition: code and dependence graph for function `vag`. By naming the independent data streams in the dependence graph, it can be found that this function computes the vector expression \( Z[i] = X[Y[i]] \). This is a sparse memcpy.

Another case corresponds to a code where the dependence graph has no memory dependence, and MAQAO could build a vector expression corresponding to the computation. In the case presented in Figure 19, this is a DAXPY operation

```c
for (int i = 0; i < LEN; i++)
    a[i] += b[i] * c[i];
```

Figure 19: Idiom recognition: code and dependence graph for function `vpvtv`. By naming the independent data streams in the dependence graph, it can be found that this function computes the vector expression \( Z = Z + X \times Y \).

**Limits** Most of the limitations come directly from the dependence analysis itself. Some remaining limits are part of identifying the correct vectorization transformation.
Structurally, MAQAO focuses on loops. When the loops are fully unrolled, or when the loop bodies are part of other functions (not inlined), MAQAO cannot detect vectorization opportunities.

More generally, loop rerolling (even after a partial unroll) would be difficult to advocate for. Indeed, it requires to identify that different slices of computation are equivalent and can be “factorized” in a loop. This rerolling comes with data reshaping issues and is not considered so far.

5 Performance Analysis

The evaluation of our method as been conducted on TSVC benchmark suite[7] and several codes from the MontBlanc project. The objective is different on both cases: TSVC indeed illustrates the capacity of MAQAO to detect a number of optimization opportunities, while for the real application, the goal is to guide optimization with MAQAO and obtain better performance results.

5.1 TSVC benchmark

MAQAO performs a dependence analysis on the code in order to find conditions for vectorization. For ARM, the dependence graph is only based on register dependences. On contrary to the x86_64 implementation, the trace of all memory references is not collected and memory dependences are therefore not computed. This implies that some loops that may not be vectorizable, due to memory dependence, may be found as vectorizable. The instrumenter is not yet connected to the memory trace library NLR [6] but this will be the focus of the following developments. However, as for the analysis for x86_64, we generate hints corresponding to vectorization opportunities. The method is similar to the method exposed in the paper [8].

We present in the following the list of hints automatically generated.

5.1.1 Reductions

Reductions correspond to cycles in the dependence graph, using an associative operation (such as ADD, MAX, MIN). If the weight of the cycle is lower than the size of a vector, then this is a reduction (otherwise the code is vectorizable). Reductions are vectorizable but require some additional transformation and rescheduling.

The table in Figure 20 shows the reductions found on TSVC benchmark. Most functions used are either vadd or vmla. The latest corresponds to a dot product.

5.1.2 Large Memory Strides

Memory strides are detected statically thanks to the reduction computing the address register used by load and store instructions. For instance, function s1115 in Figure 21 exhibits one load access with a 1024-byte stride, due to array c. This appears in its dependence graph with a reduction (add) with 1024 strides. The hint generated is the following:

- memory stride analysis - checking whether data strides are compatible with vectorization
  1.39 has references with large or negative index strides
  > memory load with stride of 1024 bytes
### 5.4.1 Complex Control Flow

When several execution paths are present in the innermost loop of a function, it makes vectorization more difficult. Using versioning, loop splitting or conditional masks (or conditional moves for ARM) is required in general.

Table 22 provides the list of functions with complex control flow. This is interesting to see that for a number of functions with conditionals, the compiler is able to flatten the conditional and generate code with conditional moves (such as s274). The most complex case corresponds to a switch statement.

### 5.4.2 Non vectorizable loops

The following functions have been considered as non-vectorizable: 

s321, s3111, s322, s352.

Since only dependence between registers are taken into account on ARM, it implies that there is a dependence cycle involving more than one instruction. These codes may correspond to scans.
for (int i = 0; i < LEN2; i++)
    for (int j = 0; j < LEN2; j++)
        aa[i][j] = aa[i][j]*cc[j][i] + bb[i][j];

Figure 21: s1115 function with its dependence graph. One of the induction register, r1, has an
increment of 1024. This corresponds to a large stride in the vector cc.

(computing partial sums) that could be vectorizable (s321), codes with flattened conditionals
(s3111) or code mixing different instructions for addition (vadd and vmla such as s352) that
could be vectorizable.

5.2 Porting and Optimizing SMMP with MAQAO

SIMPLE MOLECULAR MECHANICS FOR PROTEINS (SMMP) is an application proposed
by the University of Oklahoma, Leibniz Institute for Molecular Pharmacology, Academia Sinica
and Juelich Supercomputing Centre. SMMP proposes algorithms of Monte-Carlo type in order
to simulate the thermodynamics of proteins. The code is written in Fortran and several parallel
versions are available. However it was not ported yet to ARM architecture.

<table>
<thead>
<tr>
<th>Function</th>
<th># of execution paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>s253</td>
<td>2</td>
</tr>
<tr>
<td>s272</td>
<td>2</td>
</tr>
<tr>
<td>s277</td>
<td>2</td>
</tr>
<tr>
<td>s278</td>
<td>2</td>
</tr>
<tr>
<td>s1279</td>
<td>2</td>
</tr>
<tr>
<td>s279</td>
<td>2</td>
</tr>
<tr>
<td>s441</td>
<td>2</td>
</tr>
<tr>
<td>s442</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 22: List of functions with complex control flow. Two paths means there is an ’if’
statement that has not been flattened by the compiler. More than 2 paths implies a switch
statement for instance.
5.2.1 Porting and profiling

The main limitation of our benchmark platform (the ODROID-XU+E, ARM32 processors) is that it is only able to perform single precision computation. Changing this in the code still leads to an execution with no error and the first results are correct. The execution time does not exceed one hour and is therefore appropriate for performance tuning. To compile and vectorize the code, the following flags are used: \texttt{-O2 -g -funsafe-math-optimizations -ffree-vectorize -mfpu=neon -ftree-vectorizer-verbose=4}.

This application is proposed with 5 examples, 3 will be used in this study:

- annealing
- multicanonical
- parallel_tempering

The two other examples are not considered: one has issues related to single precision (not converging), and the second has a too small execution time to expect any interesting performance gain.

A basic profiling is conducted with gprof, in order to focus our study only on hot functions. For the 3 considered examples, the profiling graphs are shown in Figures 23, 24 and 25. The exclusive time spent in each function is the value given between parenthesis.

The hot spots of these examples are the two functions \texttt{enyshe} and \texttt{enysol} taking resp. 93.38\% and 77.58\% of the total exclusive time (only spent in the function itself). Most of the time is spent in loop nests. Figure 26 shows one of the innermost loops of function \texttt{enysol}. Note that the accesses to \texttt{xyz} and \texttt{spoint} do not allow vectorization. The number of iteration count for this loop is small, but the loop is executed many times.

5.2.2 MAQAO Analysis

The SIMD report of MAQAO for \texttt{enysol} function is not provided in extent, but we highlight some interesting points in this section. We compare the output of the optimization report given by GCC with the hints given by MAQAO. Note that at this point MAQAO is only performing a static dependence analysis, hence some memory dependence, possibly preventing SIMDization, are not taken into account. MAQAO provides “optimistic” hints. For instance Figure 27 presents a loop given as SIMDizable by MAQAO while the memory dependence prevents this. This will be corrected with trace-based dependences (see instrumentation). On the contrary, the loop presented in Figure 28 cannot be vectorizable according to GCC, and is detected as vectorizable by MAQAO. The GCC optimization report indicates that booleans have a type \texttt{LOGICAL(kind=4)} that prevents the compiler from vectorizing. A number of initialization loops and computation loops involving these booleans cannot be therefore vectorized. A possible modification is therefore to change the type of these variables (as in C where integers encode booleans) and observe the impact on vectorization. On loops 650 and 653 presented in Figure 29, MAQAO indicates that a 16000 byte stride exists between elements. The concerned optimization report is given in Figure 30. This stride comes from the way elements of arrays \texttt{spoint} and \texttt{xyz} are iterated. To get rid of this stride, there are two possibilities:

- Loop interchange,
- Transpose the arrays.

The MAQAO report shows that loop 660 has 8 possible execution paths. For inner loops, multiple execution paths prevent the compiler from aggressively optimizing and vectorizing. The
The dependence analysis provided by MAQAO shows that there is no dependence between instructions in the different execution paths, as shown in Figure 31. We could therefore split this loop into at least 3 different loops, and this could help the compiler to vectorize. This also loop exhibits stride issues, since elements are not read consecutively.

However, even when considering these transformations, there still remains conditional instructions in loop bodies, and this prevents many optimizations. Loops 645 and 655 have resp. 2 and 3 possible execution paths. Loop 655 has 2 nested conditionals. These conditions are interdependent and cannot be modified. The two paths for loop 645 result from one conditional, depending on an indirect memory access as can be seen in Figure 33. A possible solution would be to copy first the values of $rvdw(indsort(ii))$ in a new array, outside of the loops, so that the compiler could vectorize the code (using a mask).

5.2.3 Optimizing for SIMDization

In order to improve the SIMDization by the compiler, the following optimizations have been performed on the code of the $enysol$ function.

Booleans have been modified into integers for all the code with the following condition (as in C)
• TRUE = 1
• FALSE = 0

All conditions have been rewritten to preserve code correctness. This change (difficult to do automatically) improved significantly the capacity of GCC to vectorize loops.

Loop 660 is split into 3 loops in order to reduce its complexity. However gcc was still not able to vectorize. This modification has another beneficial impact: as the loop accesses 4 large arrays (16000 bytes), reducing the number of accesses to different arrays reduces the possibility to have cache conflicts.

Several optimization have been performed on loops 650 and 653. Loop interchange is not possible due to memory dependences. Transposition is still possible, requiring a copy-in outside of the loop for $xyz$ and $spoint$ arrays. This solution was tested but the gains are nullified by the time taken by the copy-in. Another possibility is to change from the beginning the data layout of these arrays, so that no additional copy is required. A more global analysis shows that $xyz$ and $spoint$ are only used in this function. The initialization was therefore modified so as to transpose all accesses. The modified loops are shown in Figure 34.

After all optimizations have been applied, compilation is achieved with the flags `-O2 -g -funsafe-math-optimizations -ftree-vectorize -mfpu=neon -ftree-vectorizer-verbose=4`. The GCC
D5.4 - MAQAO ARM Port
Version 1.1

lst=1
do il=1,npnt
  sdd=0.0
  do ilk=1,3
    sdd=sdd+(xyz(lst,ilk)+spoint(il,ilk))**2
  end do
end do
...

Figure 26: Loop example from enysol

do i=1,ncbox
  inbox(i+1)=inbox(i+1)+inbox(i)
end do

Figure 27: Sequential loop given as vectorizable by MAQAO, enysol function

optimization report indicates that loops 650 and 653 have been vectorized, together with the initialization loops. After verification on the assembly code, vectorization has been achieved indeed.

In order to improve performance, initialization loops have been replaced by calls to memset in enysol.

We compare the impact of these optimizations on performance and energy. Both code versions are compiled with the same flags: `-O2 -g -funsafe-math-optimizations -ftree-vectorize -mfpu=neon`. The first version has no hand-tuned optimizations while the second one corresponds to the version with improved vectorization and memset. The first version executes in 55min39s and the optimized version in 34min9s, exhibiting a speedup of 1.6.

Figure 35 represents energy measurements on both versions of SMMP, with parallel_tempering_s. Figure 35 on the left, shows instantaneous consumption in the two versions of SMMP; there is no difference between the two versions. The additional energy cost due to the SIMD pipeline, observed on TSVC benchmarks, has been smoothed out here. Figure 35 on the right, shows the total energy for the execution of SMMP, optimized and not optimized. The optimization brings a factor of 1.5. This shows mainly the impact of vectorization on energy consumption.

5.3 Single Precision Limitations for Other Applications

The study of applications such as Profasi, Quantum Espresso or BigDFT has led to some difficulties on the single-precision architecture considered.

Profasi The code considered has not yet been ported to ARM architecture. The code is a protein folding and aggregation simulator, developed at Juelich Supercomputing Center. This application relies on an iterative solver. By transforming data into single precision, the time for convergence increased dramatically (from 40 min to 8 days).

do il=1,npnt
  surfc(il)=.false.
end do

Figure 28: Initialization loop of an array of booleans, enysol function
lst=1
do il=1,npnt
  sdd=0.0
  do ilk=1,3 650
    sdd=sdd+(xyz(lst,ilk)+spoint(il,ilk))**2
  end do
  if(sdd.gt.radv2(lst)) then
    do ik=1, nnei
      sdd=0.0
      do ilk=1,3 653
        sdd=sdd + (xyz(ik,ilk)+spoint(il,ilk))**2
      end do
      ...
end if
end do

Figure 29: Loop 650 and 653 of enysol

- memory stride analysis
- checking whether data strides are compatible with vectorization
  - 1.650 has references with large or negative index strides
  - memory load with stride of 16000 bytes
- memory stride analysis - checking whether data strides are compatible with vectorization
  - 1.653 has references with large or negative index strides
  - memory load with stride of 16000 bytes

Figure 30: Report fragment of MAQAO for loops 650 and 653

Quantum Espresso The code is developed and used by Cineca, Democritos National Simulation Center and University Pierre and Marie Curie. This code studies the structure of material at the nanoscopic scale. Transforming this code from double precision to single precision generates approximations: atoms overlap due to the lack of precision. The application detect errors and stops.

BigDFT This application is developed at the CEA of Grenoble and by the Insitut Nanoscience et Cryogenie (INAC). This code has been developed and optimized for ARM. However, the optimization considered is not on vectorization and the code works in double precision. When simplified, the code corresponds to a stencil pattern. The optimization of the code relies not only on the SIMDization but also on the reuse of data elements [10]. This requires the trace-based analysis we are currently porting to ARM.

6 On-going Developments

The on-going developments on MAQAO are the following:

- Porting MAQAO on ARM64: this is an extension of what has been done on ARM32, as the libopcode (disassembling routines for objdump, used in MAQAO) for ARM64 relies on what is written for ARM32.
do j=nlow+1,nup
    if(xat(j).le.xmin) then
        xmin=xat(j)
    else if(xat(j).ge.xmax) then
        xmax=xat(j)
    end if
    avr_x=avr_x+xat(j)

    if(yat(j).le.ymin) then
        ymin=yat(j)
    else if(yat(j).ge.ymax) then
        ymax=yat(j)
    end if
    avr_y=avr_y+yat(j)

    if(zat(j).le.zmin) then
        zmin=zat(j)
    else if(zat(j).ge.zmax) then
        zmax=zat(j)
    end if
    avr_z=avr_z+zat(j)

    if(rvdw(j).ge.rmax) rmax=rvdw(j)
end do

Figure 31: Loop 660, enysol

- control flow analysis
  . l.660 has 8 different execution paths
    > complex control flow hinders vectorization. Use versioning, loop splitting or masks

Figure 32: Report fragment of MAQAO for loop 660

do ii=inbox(jbox)+1, inbox(jbox+1)
    if(rvdw(indsort(ii)).gt.0.0) then
      look(jcnt)=indsort(ii)
      jcnt+=1
    end if
end do

Figure 33: Loop 645, enysol
Figure 34: Optimized loops 650 and 653, enysol

Figure 35: Mean instantaneous consumption (left) and global consumption (right) for SMMP original and optimized codes

- Higher level instrumentation: A higher level instrumentation requires first dynamic loading of an instrumentation library. This requires a modification of the PLT ELF section, this is ongoing work. In particular, capturing memory traces will be possible as soon as this goal is achieved.

- Analysis of the data layout. This analysis requires the use of memory traces, in order to capture the full dependence graph (including memory dependences) and memory patterns. As presented in LCPC 2014 [1], this analysis aims at measuring the impact of data layout transformations.

- Integration into BOAST [9]. This corresponds to the next deliverable for MAQAO. To reach this goal, we rely on the previous analysis, altogether with the current SIMD analysis, in order to provide feedback to BOAST.

The two first items are incremental w.r.t. the current MAQAO state. The last item does not depend on the target architecture. It requires however that the memory trace is captured through instrumentation. The algorithmic part of this approach has already been worked on and the results for Intel x86 have been presented in a workshop. As soon as the instrumentation API enables memory traces, the third item will be tested on ARM.
The objective of data layout transformations is to evaluate the impact of this kind of transformations on other optimization (in particular vectorization) and evaluate the impact of reuse, in particular for BigDFT. The solutions proposed by MAQAO will not be semantically correct in the general case, since the analysis is based on some profile. However, depending on the code (if in particular the code is regular), the data layout transformations can be made more generic.

As this is already the case, performance will not be predicted through a performance model. Instead, we will resort to performance measures of the code generated. This is particularly important for data layout transformations where cache effects, coherence traffic and memory bandwidth are difficult to model and have a large impact on performance.

7 Conclusion

This report shows how MAQAO can analyze and provide some feedback to the user for ARM codes. This analysis is essentially static and concerns vectorization, while a preliminary working instrumentation of binary ARM codes has been developed in MAQAO, opening the way for static/dynamic analyses.

Several applications of the MontBlanc project have been analyzed with MAQAO on ARM32, and TSVC benchmarks have been used in order to test the different features of MAQAO. The development of instrumentation using dynamic libraries is the next step, with the extension to ARM64 codes. The method to analyze memory layouts and reuses, that has been developed in parallel, will then be tested on ARM architecture in the context of performance tuning with Boast.

References


