Performance Tuning of x86 OpenMP Codes with MAQAO

Denis BARTHOU[1], Andres CHARIF RUBIAL[2], William JALBY[2], Souad KOLIAI[2], and Cédric VALENSI[2]

[1] University of Bordeaux, LaBRI/INRIA, France
[2] University of Versailles Saint-Quentin, LRC ITACA, France

Abstract. Failing to find the best optimization sequence for a given application code can lead to compiler generated codes with poor performances or inappropriate code. It is necessary to analyze performances from the assembly generated code to improve over the compilation process. This paper presents a tool for the performance analysis of multithreaded codes (OpenMP programs support at the moment). MAQAO relies on static performance evaluation to identify compiler optimizations and assess performance of loops. It exploits static binary rewriting for reading and instrumenting object files or executables. Static binary instrumentation allows the insertion of probes at instruction level. Memory accesses can be captured to help tune the code, but such traces require to be compressed. MAQAO can analyze the results and provide hints for tuning the code. We show on some examples how this can help users improve their OpenMP applications.
Keywords : code optimization, performance analysis, static analysis, dynamic analysis, binary rewriting

1 Introduction

Modern processors rely on many complex hardware mechanisms in order to reach high levels of performance. In particular, the use of all levels of parallelism and the appropriate use of the memory hierarchy to hide large memory latencies are both required to obtain the full computing capacity of processors. This road to high performance is paved with many complex compiler optimizations, using, according to the code, prefetching mechanism, vectorization, loop transformations for better cache usage or data layout restructuring. While many optimizing compilers are able to perform all these transformations, they have a poor knowledge of the application context and must be conservative in their transformations. Failing to find the best optimization sequence for a given application code, this leads to compiler generated codes with poor performance, or with inappropriate code.

The performance tuning process therefore implies to guide the compiler, through pragmas, compilation flags, or source to source restructuring, to the generation of better code. Many approaches to performance tuning have been proposed, getting feedback from the application either by collecting execution traces through instrumentation (with Dyninst[4] or Pin [19] for single processors, with Scalasca[20] for multi-node systems) or hardware counters values (such as Intel Vtune or PTU for instance). Hardware counter-based techniques show how the architecture behaves with the considered code and input set. However, it is difficult to make the

connection between hardware event counts and source code, since both source code and compiler optimizations have an impact on the resulting hardware events. Moreover, there is no direct link between hardware counters and the quality of the compiler generated code. To have feedback from the compilation process, it is necessary to analyze performance from the assembly generated code.
In this paper, we describe how our MAQAO [6] tool (Modular Assembly Quality Analyzer and Optimizer) handles performance analysis and memory tracing for OpenMP programs. Although in this paper, our target architecture is Core2, the tool can be easily retargeted to other x86 architectures essentially by changing the performance models used. Targeting other architectures requires more work (dealing with different instruction sets) but the main principles can be adapted fairly easily: an earlier version of MAQAO was targeting IA64 architectures which are very different from X86. This tool combines static analysis of compiler-generated assembly code with the analysis of execution traces and binary instrumentation. Static performance evaluation provides hints on how to improve the compilation process, and assess the amount of performance that could be obtained through optimization. This estimation is performed on the sequential codes executed by threads. Improving unicore performance (both in sequential and parallel part of the codes) contributes to improving global performance and efficiency of the code. Dynamic, thread-wise traces, in particular compact memory traces, show how to improve interactions between threads, and detect false sharing situations, for instance. We show in particular how static performance evaluation is achieved on Core 2 architecture and how compact memory traces can be used to help tune OpenMP code performance.

2 Static Performance Evaluation

MAQAO relies on static performance evaluation to identify compiler optimizations (or lack of), patterns of codes that are not efficient, and assess performance of loops. The performance model and its use for x86 architecture is described in this section. We first recall how MAQAO analyzes and restructures codes.

2.1 Code restructuring

MAQAO exploits static binary rewriting for reading and instrumenting object files or executables. Static binary rewriting refers to the post-link time manipulation of binary executables. This approach has the advantage, compared to approaches requiring compiler interaction (analysis of assembly code) or inclusion of libraries (for heap monitoring for instance), to obviate the need of recompiling or relinking. The API for reading and manipulating static binary files is defined by MADRAS [18], a generic disassembler and instrumenter generator. MADRAS takes a grammar associating binary expressions to assembly instructions, similarly to `yacc` grammars, and generates a corresponding disassembler, using a linear-sweep method (similar to `objdump`). This disassembler for x86 is then used by MAQAO.
The disassembled binary code is restructured: call graphs and control flow graphs, loops and dependence graphs on registers are built (Fig. 2). The call graph construction uses labels found in the binary, if any. Both call and control flow graphs are limited in the presence of indirect jumps and self-rewriting codes. So far,

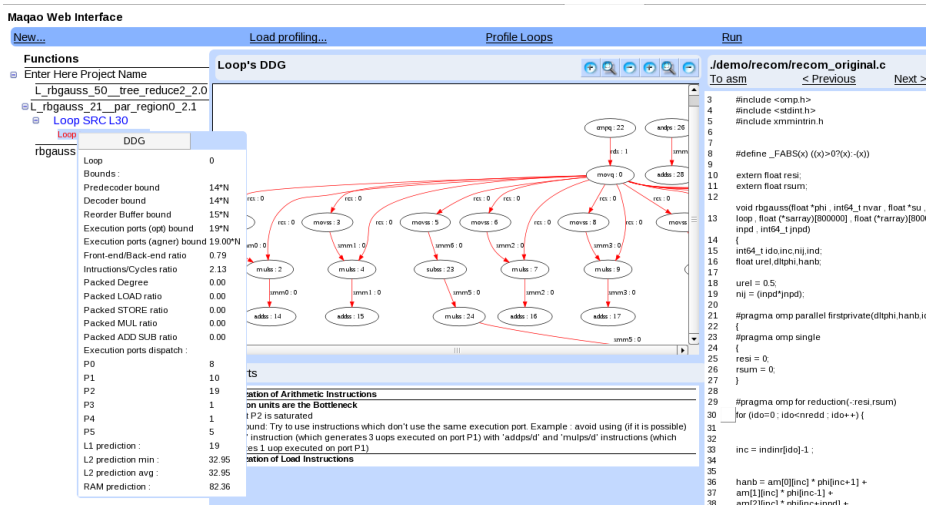


Fig. 1. The MAQAO user interface

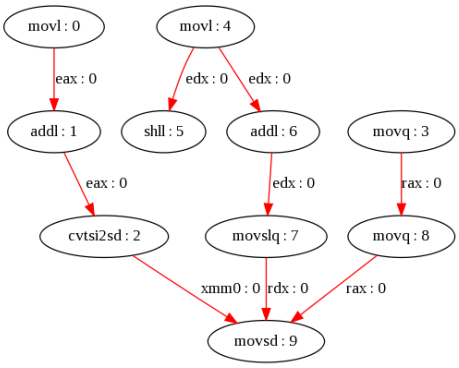


Fig. 2. Data dependency graph of a loop

there is no (partial) interpretation of the code in order to resolve indirect jumps and self-rewriting of codes. While the first limitation may prevent MAQAO from finding correct control flow, the later may lead to incorrect disassembling. Natural loops are built using a fast algorithm [9].

There is a direct link between each assembly statement and a source code statement provided the debugging information is present (usually given when compiling with `-g` flag). This link allows the detection of some compiler optimizations, such as multiple versioning, inlining and unrolling to some extent. Innermost assembly loops are grouped by source line so that users can visualize the generated assembly loops for a given source loop (Fig. 3).

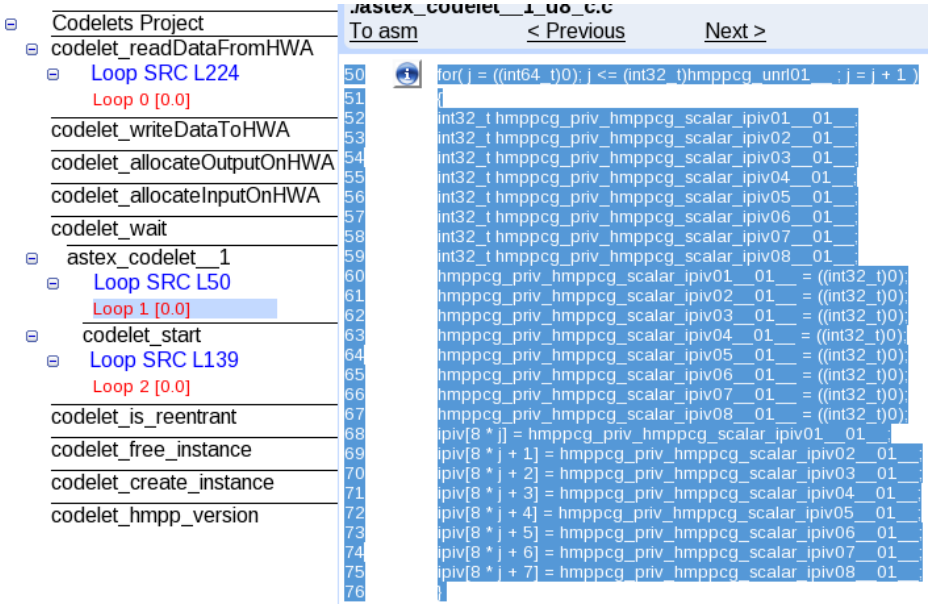


Fig. 3. Project - Files - Functions - Loops Hierarchy and corresponding source

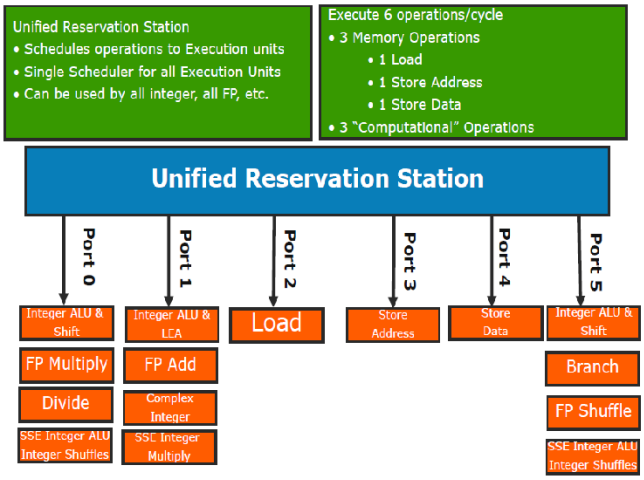
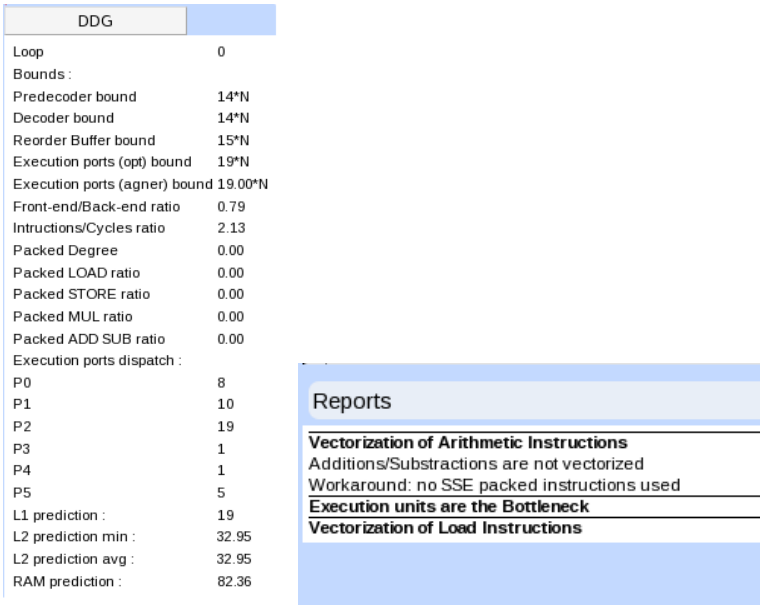


Fig. 4. Core2 execution unit overview

2.2 Performance Model

The performance model of MAQAO computes performance estimates based on the assembly code. It evaluates the cycles required for executing innermost loops. The reason for considering only the innermost loops is that they usually constitute the most time consuming part of the code. The x86 architecture model we con-



(a) MAQAO statistics (b) MAQAO reports

Fig. 5. MAQAO interface details.

sider takes into account the front-end pipeline (decoding, permanent register file allocation, special microcoded instructions), the different ports for the execution units, and the latencies of instructions. For memory instructions, several latencies are considered, according to the location of the data in memory hierarchy. For other instructions, latencies are tabulated, either coming from microbenchmarks or from Agner documentation [8]. Note that the evaluation only provides an optimistic bound, meaning that the real code may execute in more cycles due to some extra latency not taken into account by our model.

Among different metrics that MAQAO can produce, we focus on the following five key metrics:

1. **Vectorization Report Analysis:** This report, shown in Fig. 5(b), provides us with individual (load, store, add, multiply) reports on vector instruction usage: for example a vector ratio of 1 for multiply operations means that all of multiply operations have been vectorized by the compiler. This ratio is computed taking into account only floating point operations and full length packed vector operations. These metrics are essential to evaluate the quality of the vectorizing capabilities of the compiler and possibly to palliate some of its deficiencies by inserting appropriate pragmas.
2. **Execution port usage:** For each execution port (Fig. 4), MAQAO computes an estimation of the number of cycles spent on each port. Our performance estimates takes into account the special case of instructions which are split into different micro-operations to be executed on multiple ports [8]. When an instruction (or a micro operation) can be executed on different ports (a common example is simple integer instructions which can be assigned indif-

ferently to P0, P1 and P5), the less saturated port is chosen. Figure 5(a) shows the report presented by MAQAO Since all of the ports can operate in parallel, this metric is essential to measure the amount of parallelism exploitable between the key functional units: add, multiply, load and store units. This provides a first estimate of a best performance case (assuming all operands are in L1) and also of the potential imbalance between the port usage. For example, this allows to quickly detect whether a code is memory bound and to get a first quantitative estimate of how much a code is memory bound. The number of cycles spent on every port gives us an accurate ranking on the potential bottlenecks of the code difference in cycles between first order and second order bottlenecks).

3. **Performance estimation in L1:** Taking into account all of the limitations of the pipeline front end and of the pipeline back end, MAQAO provides us with an estimate of the cycles necessary to execute one loop iteration assuming all operands are in L1. The limitations that we are taking into account are: instruction predecoding, instruction decoding, permanent register file allocation, special microcoded instructions. As mentioned earlier, in most cases this bound is only useful as a lower bound.

4. **Performance estimations in L2/RAM:** Relying on memory access patterns detected at the assembly level and micro benchmarking results on the same memory patterns, MAQAO computes an estimate for the execution time of a loop iteration, assuming all operands are in a given level of the memory hierarchy (L2 or RAM) and are accessed with stride 1. The memory patterns used for the pattern matching have previously been determined by systematic hierarchical microbenchmarking: first simple "Load X" (resp. "Store Y") kernels (performing a single read stream through an array X, resp. a simple writing stream through an array Y) are measured under various conditions (unrolling, instruction used, etc ..). Then more complex patterns "Load X Store Y", "Load X Load Y", "Load X Load Y Store Z", etc ... are measured to quantify the interaction between Load streams and Stores streams. We experimentally observed that beyond 4 array streams, most of the performance measured could be deduced from simpler patterns. Therefore this simple set of patterns is used for our performance prediction [10]. The L2 estimate constitutes a reasonable performance objective while the RAM estimate is a stride 1 worst case. The drawback of both of these estimates is that they ignore the stride problem (which in RAM will be essential) and, second, that they do not take into account the mixture of hits and misses which is typical for real applications. However, it should be noted that micro benchmarking already accounts for some typical mixture of hits/miss resulting from spatial locality usage. For stride 1 memory access, micro benchmarking does not distinguish between primary misses (occurring for the first word access to a cache line) and secondary misses/hits (occurring when subsequent words in the cache line are requested), it provides an estimate of the average time for accessing a memory location in a stride 1 access mode (array stored in contiguous memory). The stride problem can be easily corrected when the memory tracing analysis is performed, because for each load/store, the striding pattern will be then determined. Then a revised more accurate L2/RAM estimate can be generated. Again incorporating this extra information enables MAQAO to produce better performance estimates.

5. **Performance projections for full vectorization:** In cases where the code is partially or not vectorized, MAQAO computes performance estimations assuming a full vectorization. This is performed by replacing the scalar operations by their vector counterparts and updating the timing estimate due to the use of these instructions. This is particularly useful to guide the optimization process and to avoid useless efforts: for example, indirect access to arrays cannot be vectorized due to the lack of vector scatter/gather instructions in the current SSE instruction sets. However, in most loops, these indirect accesses are followed by floating point operations (adds or multiplies) which could be vectorized. The MAQAO performance projection gives us quickly an estimate of whether trying to vectorize these operations will pay off or not.

2.3 Applying MAQAO to real-world Applications

To illustrate the interest of these metrics, we performed a static analysis using MAQAO on two high performance codes from the ParMA project [16]: `RECOM-AIOLOS` from RECOM, and `ITRLSOL` from Dassault-Aviation. Two code fragments are shown in Fig. 6. The Intel C and Fortran Compilers (ifort and icc v11.0) are used to generate the assembly codes analysed by MAQAO. They are also used to generate OpenMP parallel regions when appropriate and also all of the performance measurements have been carried out using these compilers.

```
DO IDO=1,NREDD
| INC   =   INDINR(IDO)
| HANB  =   AM(INC,1)*PHI(INC+1)  &
| + AM(INC,2)*PHI(INC-1)  &
| + AM(INC,3)*PHI(INC+INPD)  &
| + AM(INC,4)*PHI(INC-INPD)  &
| + AM(INC,5)*PHI(INC+NIJ)  &
| + AM(INC,6)*PHI(INC-NIJ) &
| + SU(INC)
| DLTPHI = HANB/AM(INC,7)-PHI(INC)
| PHI(INC) = PHI(INC) + DLTPHI
| RESI = RESI + ABS(DLTPHI)
| RSUM = RSUM + ABS(PHI(INC))
```

(a) `RECOM-AIOLOS` analyzed code fragment

```
DO cb=1,ncbt
| igp = isg isg = icolb(icb+1) igt = isg  igp
c$OMP  PARALLEL DO DEFAULT(NONE)
c$OMP  SHARED(igt,igp,nnbar,vecy,vecx,ompu,ompl)
c$OMP  PRIVATE(ig,e,i,j,k,l)
| DO ig=1,igt
| | e = ig + igp
| | i = nnbar(e,1)
| | j = nnbar(e,2)
cDEC$ IVDEP
| |DO k=1,ndof
cDEC$ IVDEP
| | | DO l=1,ndof
| | | | vecy(i,k) = vecy(i,k) + ompu(e,k,l)*vecx(j,l)
| | | | vecy(j,k) = vecy(j,k) + ompl(e,k,l)*vecx(i,l)
```

(b) `ITRLSOL` analyzed code fragment

Fig. 6. Two examples of codes. The `IVDEP` pragma tells the compiler to vectorize the loops.

The different execution ports P0 to P5 in the Core2 architecture correspond to (Fig. 4):

- P0-P1-P5: computation units port
- P2: memory read port
- P3-P4: memory write ports

Depending on the number of cycles spent in each port, this information allows to detect if the code is memory bound (P2, P3-P4) or compute bound (P0-P1-P5).

The 3D-combustion modeling software `RECOM-AIOLOS` is a tailored application for the mathematical modelling of industrial firing systems ranging from several hundred kW to more than 1000 MW. In-depth validation using measurements from industrial power plants, the extension of chemical reaction models and the rapid development of computer technology have made RECOM-AIOLOS a well proven and reliable tool for the prediction equations on a 10-15 million cells finite volume grid, leading to high computational demands. Originally being designed for high-performance computing on parallel vector-computers and massively parallel systems, the software has been ported to low-cost multi-core systems to expand the hardware base [17].
The most time consuming subroutine in `RECOM-AIOLOS` is `RBgauss`, which implements a red-black iterative solver. The choice of the red-black algorithm allows for easy parallelization with, for example, OpenMP. The `RBgauss` subroutine contains two loops (denoted *Red* and *Black* loop) with a communication between them using MPI. The static analysis with MAQAO is performed on the *Red* loop as both loops are the same. It gives the following values:

- Vectorization report: all the ratios of vectorization are equal to 0%. The compiler has not vectorized the loop.
- Execution units usage (format is PORT_NUMBER:CYCLES_SPENT): P0:8 / P1:10 / P2:19 / P3:1 / P4:1 / P5:4.
- L1 prediction: 19 cycles.
- L2 prediction: 28.77 cycles.
- RAM prediction: 70.66 cycles.
- Vectorization prediction (assuming data in L1): 7 cycles.

Thanks to the static analysis of MAQAO, we can notice that the code is memory bound on Core 2, since it takes 19 cycles to execute all read instructions. This corresponds to the largest number of cycles on any given port.
The memory traces achieved using MAQAO allowed to detect that there are two arrays (`AM` and `PHI`) in the code which are accessed with a stride 2 with some gaps from time to time.
Moreover, the large number of reads and the stride 2 access imply that the code is very sensitive to cache misses [12].
Since the major bottleneck for this routine is data access from RAM combined with low spatial locality (stride 2 access), various optimizing transformations are performed, but only the following has a significant impact on performance: reshaping array AM for getting rid of the stride 2 access. More precisely, the array `AM` is split into two distinct arrays still with indirect access but stride 1. This is equivalent to reshaping an array of complex numbers by splitting it into arrays, one containing the real part, the other one containing the imaginary part.
Thanks to this optimization, the cache misses are almost half what they used to be (Fig. 7(b)). Single core performance has been improved by speedups between 1.2 and 1.3 (Fig. 7(a)) thanks to this code transformation. Multicore performance has been improved by speedups between 1.3 and 1.4 (Fig. 8).

The `ITRLSOL` (ITeRative Linear SOLver) application provided by Dassault-Aviation is the linear solver kernel of AeTHER, a larger Computational Fluid Dy-

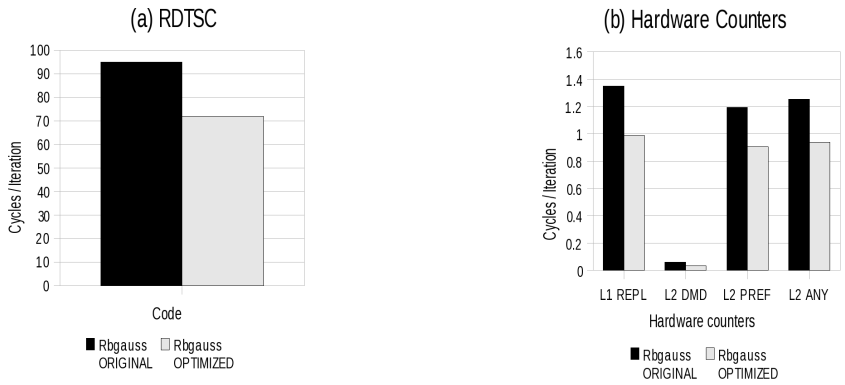


Fig. 7. `RBgauss` code optimization on unicore.

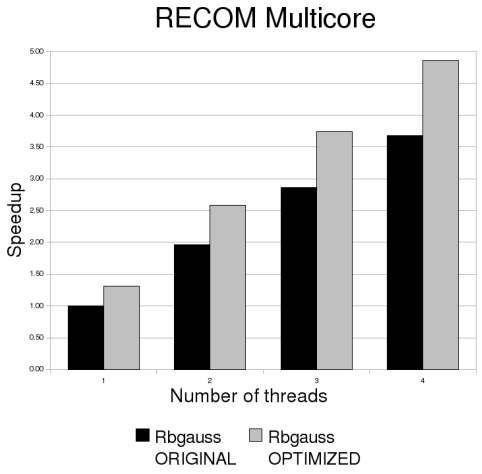


Fig. 8. `RBgauss` speedups on multicore.

namics (CFD) simulation code for the solution of Navier-Stokes equations, discretized on unstructured meshes. The most time-consuming subroutine in `ITRLSOL` is `EUFLUXm`, which implements a sparse matrix-vector product. The `EUFLUXm` subroutine contains two groups of quadruply nested loops (2 identical quadruply nested loops in each group). For the considered 4-level loop nest in this code, the report provides the following information:

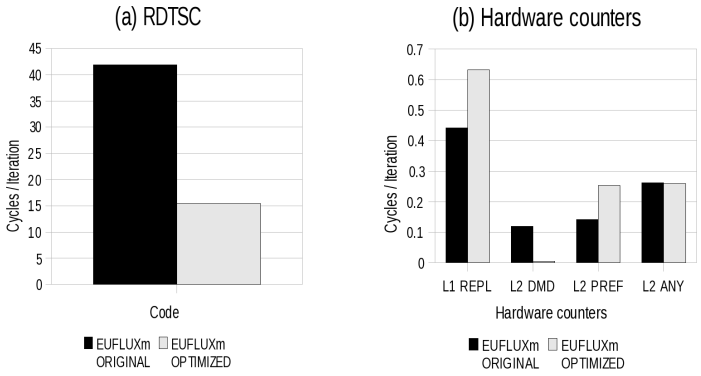


Fig. 9. `EUFLUXm` code optimization on unicore.

- Vectorization report: all the ratios of vetorization are equal to 0%. The compiler has not vectorized any loop, despite the presence of pragmas.
- Execution units usage (format is PORT_NUMBER:CYCLES_SPENT): P0:3 / P1:3 / P2:6 / P3:2 / P4:2 / P5:3
- L1 prediction: 6 cycles.
- L2 prediction: 9.08 cycles.
- RAM prediction: 37.04 cycles.
- Vectorization prediction (assuming data in L1): 3 cycles.

The static analysis with MAQAO shows that the code is dominated by memory accesses. The memory traces achieved with MAQAO allow us to detect that the inner most loops are accessing the arrays in the wrong dimension which leads to a poor spatial locality [12].
To improve the spatial locality, a transformation is done by interchanging the second loop on *ig* and the two innermost loops (the *ig* loop becomes the innermost loop). All of the arrays are now accessed column-wise. This optimization improves sequential performance by speedup of 2.5 (Fig. 9(a), Fig. 9(b)).
In a multicore environment the same optimization is applied. It gives a speedup of up to 2.5 (Fig. 10).
Thanks to the information collected from the static analysis with MAQAO, we detect that both applications `RECOM-AIOLOS` and `ITRLSOL` are not vectorized and memory bound. Using this information and applying MAQAO memory traces and PTU [3] (for performance tuning) allows us to find the performance bottleneck (stride 2 access for `RECOM-AIOLOS` and poor spatial locality for `ITRLSOL`) in these codes.

3 Memory Traces for OpenMP Codes

Memory traces represent information of crucial importance for performance tuning of multithreaded codes. Indeed, traces can help detect important inefficiencies (false sharing) or opportunities for optimizations (setting thread affinity according to reuse among threads). The major issue of memory traces is the amount of

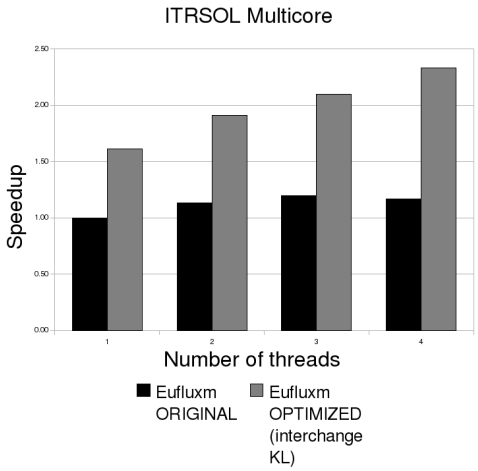


Fig. 10. `ITRLSOL` speedups on multicore.

data they represent. We first describe how the tracing is achieved in MAQAO, which algorithm we use to compress the traces and how they are used in order to tune performance.

3.1 Static Binary Instrumentation

The static binary instrumentation is achieved using MADRAS [18]. It allows instruction level instrumentation, inserting probes either provided by MAQAO (for iteration counts) or user-defined ones in libraries.
Figure 11 shows how easy it is to use this API to build an instrumenter module. The two *for* loops walk through all blocks and all instructions of the loop with id 1. For each load and store instruction, the `mt_store` function is called from the `libmaqaotrace` library which contains the implementation of the trace compression algorithm mentioned earlier taking into account multithreading. This function builds a compact trace of memory accesses. MADRAS performs instrumentation statically, through binary rewriting, allowing the instrumented program to be run without additional overhead.

3.2 Memory Traces

Memory accesses can dramatically slow down the execution time of a program, particularly when it is memory bound. Capturing the memory behavior of a program can help tune the code, using prefetching or transforming the code for a better reuse of data. However, tracing memory accesses (load, store, prefetch) by simply dumping all address streams would lead to many Terabytes of data on

```
loopid = 1;
instru = madras.new(binfilename);
instru:modifs_init();
for block in loop_blocks(loop_table[pname][loopid])do
    for ins in instructions(block) do
        if( (ins:is_load() or ins:is_store())) then
            instru:fctcall_new("mt_store","libmaqaotrace.so",ins:get_address(),0);
            instru:fctcall_addparam_imm(instru_count);
            if(ins:is_load()) then
                instru:fctcall_addparam_frominsn(ins:get_operand_src_index()-1,
                                                 ins:get_address());
            else
                instru:fctcall_addparam_frominsn(ins:get_operand_dest_index()-1,
                                                 ins:get_address());
            end
            instru_count = instru_count + 1;
            instru_tab[ins:get_address()] = instru_count;
        end
    end
end
instru:modifs_commit(binfilename.."_".."instrumented");
instru:terminate();
```

Fig. 11. MADRAS API available through MAQAO (LUA scripting interface)

real applications. The memory space for these traces is a major concern in every trace-profiling application. We first detail the compression algorithm used in MAQAO and then describe how this method has been adapted to MAQAO for tracing multithreaded codes (OpenMP programs support at the moment).

Compression algorithm The compression is ensured by an on-the-fly incremental algorithm called loop nested recognition and developed by Ketterlin and Clauss [11]. We recall in this section the main steps of this method.
Their technique represents memory address streams as union of Z-polytopes which are represented by (nested) loops. The idea of using loops to characterize an accessed region has first been introduced by Elnozahy [7]. Simpler representations have been proposed using triplets (starting addresses, stride, number of references) and their extension to multidimensional triplets [13]. This is a natural approach since the majority of time execution of a program is spent in loops, and memory accesses are regular. Figure 12 shows the parallel between a typical example of program loop and its representation.

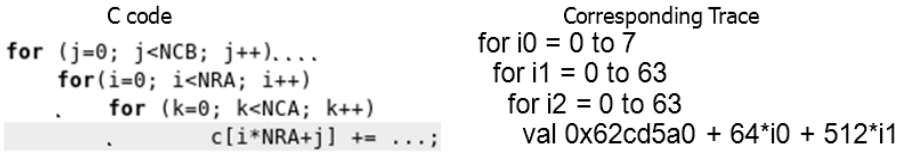


Fig. 12. Source code loop and its corresponding nested loop representation

The algorithm takes into account two types of access patterns:

- regular patterns obtained by regular or irregular accesses
- irregular patterns due to random accesses. There is no easy way to deal with this kind of pattern. Existing approaches fall back on lossy algorithms.

Each memory stream is assigned an internal stack that stores either regular and irregular patterns. Regular patterns are stored in the loop format described above. Irregular patterns, which correspond to a sequence of numbers without any affinity, are kept as it is. The stack size management is controlled by three factors:

- the maximum stack size (length)
- the maximum number of terms within the loop body representation (breadth)
- the number of elements to throw when the size limit is reached

The algorithm is lossless as long as the stack is large enough to store all memory streams. On some huge programs it may be necessary to voluntarily limit the stack in order to prevent consuming all the available memory. In this case the algorithm is lossy.

Multithread and Performance Issues We have adapted the previous method to a multicore execution context and extended it by taking into account static analysis information. Adaptation to multicore execution boils down to reimplementing the original method as a thread-safe method. Traces are saved for each memory access, for each thread independently.

Instrumenting a code for memory traces usually generates a large overhead, and most methods (such as Metric[13]) use sampling in order to reduce this weight on the execution time. Another approach is to use static analysis in order to infer fragments of traces from the assembly code. Indeed, tracking down the induction variable of inner loops makes it possible to capture the stride of memory streams. The value may appear in the code as a numerical constant or as a parametric constant (invariant in the loop). But in both cases we only need this value and the iteration count to extract the loop representation (usually found in some register). Thus this saves a large part of the overhead due to instrumentation.

Our design still suffers from a lack of information about temporal locality. This could be alleviated by using a simplified cache simulator. Its integration in MAQAO is left for future work.

3.3 Using Traces for OpenMP Performance Issues

Once trace collection is done, the results are analyzed (manually at the moment) and some hints are provided to the user to help tune the code. We provide thereafter a number of scenarii reflecting performance issues that can be detected using this trace framework.

- Potential bank/load store queue conflicts: this type of conflicts can be easily detected by comparing addresses accessed by ”neighbor” instructions. On Xeon architecture a store on address A followed by a load on address B can generate pipeline stalls if address A and B have the same low order 12 bits (same offset within a page). The performance impact will depend upon the execution distance (how many cycles apart) between the load and store instructions. Detection of this intra-thread issue consists in finding successive load/store patterns accessing different addresses sharing the same low order 12 bits.

– False Sharing: Two threads share some cache line, while they do not share any data. However, performance is impacted due to cache coherency issues. Patterns that lead to false sharing can be tracked down by comparing addresses read and written by different threads (loads and stores). Coherence issues increase with the number of cores and the memory access is not uniform (case of Intel architectures).
– Prefetch distance: Prefetch distances can be found or guessed based on the accessed regions of memory. The memory region found by our trace mechanism helps the user to determine if prefetch causes potential false sharing issues, depending on the prefetch distance.
– OpenMP work distribution scheme: Based on the memory pattern accesses, we can recover OpenMP work distribution scheme (different static, dynamic and guided modes) and evaluate which mode is the more appropriate for the application;
– Reuse degree between loads: Multiple loads on the same, shared data give us temporal locality hints. The user could, if possible, reorder some statements to take advantage of cached data at some point;
– Strided accesses: Depending on the programming language, data is stored by column or by row in memory. One possible optimization is to assess which configuration is the most efficient. Moreover, structure of arrays or arrays of structures are usual choices that impact performance, in particular due to vectorization. Evaluation from traces of the opportunity to vectorize memory accesses is an important task in the code tuning phase.

To illustrate one of these scenarii, consider the code shown in Fig. 12. This code could match for instance to a matrix multiplication code. The i loop can be parallelized with OpenMP, and different load balancing methods can be chosen, among which `STATIC` and `DYNAMIC` methods. Tracing memory writes reveals that with a `STATIC` load balancing method and 8 threads, there is no false sharing occurring. For the `DYNAMIC` method, as shown in Fig. 13, some false sharing occurs, resulting in increased memory latency due to cache coherency mechanism (false sharing for write accesses).

```
Thread 1
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255b0 + 1024*i0

Thread 2
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255b8 + 1024*i0

Thread 3
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255c0 + 1024*i0

Thread 4
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255d8 + 1024*i0
Thread 5
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255d0 + 1024*i0
Thread 6
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255a8 + 1024*i0

Thread 7
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255c8 + 1024*i0

Thread 8
for i0 = 0 to 127
  for i1 = 0 to 127
    val 0x45255a0 + 1024*i0
```

Fig. 13. Partial traces corresponding to memory writes in a DGEMM code, where the i loop is parallelized with `DYNAMIC` load balancing strategy. This solution is not efficient due to the false sharing between threads (for instance, threads 1 and 2 access the same cache line).

4 Related Work

There are few performance tools dealing with parallel (multithreaded) codes optimization.

Intel VTune [5] relies on the Thread Profiler application to determine the number of cores that are being used, show the distribution of work to threads but does not take into account memory accesses.

Acumem [1] rely on cache related statistics to predict performance bottlenecks. MAQAO uses (memory accesses) tracing rather than sampling in order to provide the user with very accurate results and detect unusual behaviours.

HPCToolkit [2] also works at binary level for language independence, collects and correlates multiple performance metric, computes derived metrics to aid analysis. However it uses profiling rather than adding code instrumentation. MAQAO supports code instrumentation to enable users inserting probes and concentrate on specific parts of an application.

PIN [19] and DynInst [4] are two tools allowing modification of an executable for the purpose of instrumentation. Both perform dynamic instrumentation, operating on the executable while it is loaded in memory and running.

PIN traces an executable during its execution (acting as a "just-in-time" compiler) and monitors various parameters. It allows to transfer control flow to external functions, effectively inserting calls to these functions, and to modify the memory, all of this while the executable is being executed. It is also possible with PIN to insert probes in the executable while it is loaded in memory but not yet running, which actually redirects the execution flow to another function. This mode does not work on multi-threaded applications and does not check the destinations of jump instructions. PIN is also able to perform some static analysis of a file (like identifying functions arguments).

DynInst [4] allows dynamic updating of code, a process labeled runtime injection. It proceeds by directly updating a program in memory to insert jumps pointing to the added sections of code which reside somewhere else in memory. A recent update also allows DynInst to perform some binary rewriting.

Instrumentation by MADRAS is not accomplished at runtime by another thread, as it is performed statically. The instrumented program can then be run without additional overhead but the calls to the instrumented functions. No special environment is required.

The integration of the MADRAS library allowed us to introduce the memory tracing feature.

Valgrind [14] is a dynamic binary instrumentation and analysis framework which uses a simulated CPU to analyse programs (in particular on cache and memory use) and offer instrumentation options. The simulated CPU causes an important slowdown of the analysed program and requires more memory space.

METRIC [13] uses dynamic instrumentation to capture memory accesses and scope changes.

PSnAP [15] also uses dynamic instrumentation to generate memory stream profiles on a per loop basis as MAQAO does.

Ketterlin & Clauss [11] propose a more sophisticated compression technique that we are using in our memory tracing library.

To our best knowledge, there is no existing technique for memory tracing of parallel (multithreaded) codes.

5 Conclusions and future work

MAQAO is a tool for performance tuning that relies on both static analysis of binaries and on data collected through instrumentation. We have shown in this paper how the performance model for x86 processors is designed inside MAQAO and how memory tracing for OpenMP programs is achieved.
The static analysis is combined with the hint mechanisms of MAQAO, helping the user to locate easily in the application source code the code fragments that exhibit poor performance. Moreover, this analysis provides a rough estimate of the possible performance gains that could be expected by an efficient vectorization.
The memory tracing method we propose relies on two mechanisms: a new binary instrumentation framework, MADRAS, where each assembly instruction can be instrumented individually, and a compact memory trace representation [11], extended for multithreaded programs. We have shown, through multiple scenarii, how the multithreaded trace information can be used to detect performance issues specific to multicore machines.
For future work, we plan to improve the trace representation in order to capture partially some scheduling information (associating time stamps with memory addresses). In future versions, trace results will be analysed automatically. MAQAO still needs the assembly code for building its analysis and will rely only on the data extracted from the disassembled binary in the next release (the disassembled binary from MADRAS being used in correlation with it to retrieve the instructions addresses).

References

1. Acumum AB. Acumem SlowSpotter and Acumem ThreadSpotter, 2009. http://www.acumem.com/content/view/133/182/.
2. 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. Technical Report TR08-06, Rice University, 2008.
3. A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization Made Easier with Intel Performance-Tuning Utility, 2007. http://www.intel.com/technology/itj/2007/v11i4/.
4. B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Intl. Journal of High Performance Computing Applications*, 14:317–329, 2000.
5. Intel Corporation. Intel VTune Performance Analyzer 9.1, 2009. http://software.intel.com/en-us/intel-vtune/.
6. L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J-T. Acquaviva, and W. Jalby. Exploring Application Performance: a New Tool For a Static/-Dynamic Approach. In *Los Alamos Computer Science Institute Symp.*, Santa Fe, NM, October 2005.
7. E. N. Elnozahy. Address trace compression through loop detection and reduction. *SIGMETRICS Perform. Eval. Rev.*, 27(1):214–215, 1999.
8. Agner F. Software optimization resources, 2009. http://www.agner.org/optimize/.
9. L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. *Algorithms - ESA*, 3221:677–688, 2004.

10. W. Jalby, C. Lemuet, and X. Le Pasteur. A New Set of Microbenchmarks to Explore Memory System Performance for Scientific Computing, 2004. International Journal of High Performance Computing Applications.
11. A. Ketterlin and Ph. Clauss. Prediction and Trace Compression of Data Access trough Nested Loop Recognition. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, 2008.
12. S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby. A Balanced Approach to Application Performance Tuning. In *Proc. of LCPC*, LNCS, Delaware, USA, October 2009. Springer.
13. J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting. *ACM/IEEE Int. Symp. on Code Optimization and Generation*, 0:289, 2003.
14. N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. 2007. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.
15. C. Mills Olschanowsky, M. Tikir, L. Carrington, and A. Snavely. PSnAP: Accurate Synthetic Address Streams Through Memory Profiles. In *Int. Workshop on Languages and Compilers for Parallel Computing*, 2009.
16. ParMA ITEA2 Project: Parallel Programming for Multicore Architectures. http://www.parma-itea2.org/.
17. B. Risio, A. Berreth, S. Zuckerman, S. Koliai, M. Ivascot, W. Jalby, B. Krammer, B. Mohr, and T. William. How to Accelerate an Application: a Practical Case Study in Combustion Modelling. In *Proc. of ParCo*, Lyon, France, 2009.
18. C. Valensi and D. Barthou. MADRAS: Multi-Architecture Disassembler and Reassembler, 2009. http://maqao.prism.uvsq.fr/wiki/wiki/MadrasDownload.
19. S. Wallace and K. Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 209–217, San Jose, CA, March 2007.
20. F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proc. of the 2nd HLRS Parallel Tools Workshop*, pages 157–167, Stuttgart, Germany, July 2008. Springer. ISBN 978-3-540-68561-6.