An Approach to Application Performance Tuning

Andres CHARIF-RUBIAL [a], Souad KOLIAI [a], Stéphane ZUCKERMAN [a], Bettina KRAMMER [a], William JALBY [a] and Quang DINH [b]

[a] *University of Versailles Saint-Quentin-en-Yvelines, France*
[b] *Dassault-Aviation, France*

Abstract. Current hardware trends place increasing pressure on programmers and tools to optimize scientific code. Numerous tools and techniques exist, but no single tool is a panacea; instead, an assortment of performance tuning utilities are necessary to best utilize scarce resources (e.g., bandwidth, functional units, cache). This paper describes an optimization strategy combining static assembly analysis using the MAQAO tool with dynamic information from hardware performance monitoring (HPM) and memory traces. A new technique, decremental analysis (DECAN), is introduced to iteratively identify the individual instructions causing performance bottlenecks. We present a case study on an industrial application from Dassault-Aviation on a Xeon Core 2 platform. Our strategy helps discover and fix problems related to memory access locality and loop unrolling, which leads to a sequential and parallel speedup of up to 2.5.

Keywords. code optimization, performance analysis, static analysis, dynamic analysis

1. Introduction

In High Performance Computing (HPC), there is a constant hunger for more resources (e.g., CPU, RAM, I/O). With finite limits on these resources, it is the responsibility of the programmer, interacting with the compiler, to optimize an application for peak performance. Optimization consists of gathering data about a program's behavior, diagnosing the problem by identifying resources that are saturated and the instructions at fault, and prescribing a solution which entails applying a change to the code's algorithm, structure, or data layout.

The first step, data collection, involves an array of different analysis techniques to examine different aspects of application performance. Typically, a code is deemed optimal if it approaches the peak numerical throughput of the processor; this implies that only an algorithmic change could further improve performance.

In this paper, we address optimization of HPC applications, specifically CPU- and memory-bound ones. We describe a semi-automated methodology to analyze performance and guide the optimization process. Both static analysis (with MAQAO and visual inspection) and dynamic analysis (of memory access patterns) of the code are performed. Information from these analyses guides us to the regions of code furthest from peak performance. Using this information, we introduce a new approach to identify the

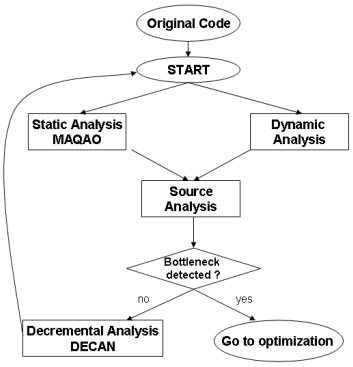

Figure 1. Methodology diagram

specific set of instructions that are responsible for increased computation latency: decremental analysis (DECAN). DECAN involves systematically changing instructions' behavior in a particular region to identify the runtime contribution of each instruction or set of instructions. DECAN is useful in those cases where the static and dynamic analyses do not give enough information about the program's behavior.

This methodology has been applied to several industrial HPC codes, among them `ITRLSOL` (ITeRative Linear SOLver) from Dassault-Aviation, which is presented in this paper and where we achieved a speedup of nearly 2.5.

Section 2 presents the tools and techniques in our analysis approach. Section 3 describes a case study (`ITRLSOL`) on which our methodology was applied leading to significant performance improvements. Section 4 discusses other approaches to performance tuning. Finally, Section 5 concludes.

2. Toward a better evaluation process

This section provides a high-level description of each step of the methodology. Figure 1 shows a flow diagram of the process with the following performance analysis techniques being intended to identify the root cause of a performance bottleneck. It is assumed that the targets of optimization are significant contributors to program execution time as reported by tools such as `gprof` [7] or Intel PTU [1].

2.1. Static Analysis: MAQAO

MAQAO [5] is a static analysis tool which aims at analyzing assembly code produced by the compiler, extracting key characteristics from it and detecting potential inefficiencies. MAQAO was originally developed for the Intel Itanium architecture, and was recently extended to support x86 programs using Intel Core 2 processors as well. Figure 2 shows the graphical interface for browsing compiled programs. It can provide the user with many metrics derived from the assembly code analysis:

1. **Vectorization Report Analysis:** provides individual measures on vector instruction usage (load, store, add, multiply). These metrics are essential to evaluate the quality of the vectorizing capabilities of the compiler and therefore try to palliate some of its deficiencies by inserting appropriate pragmas or directives.

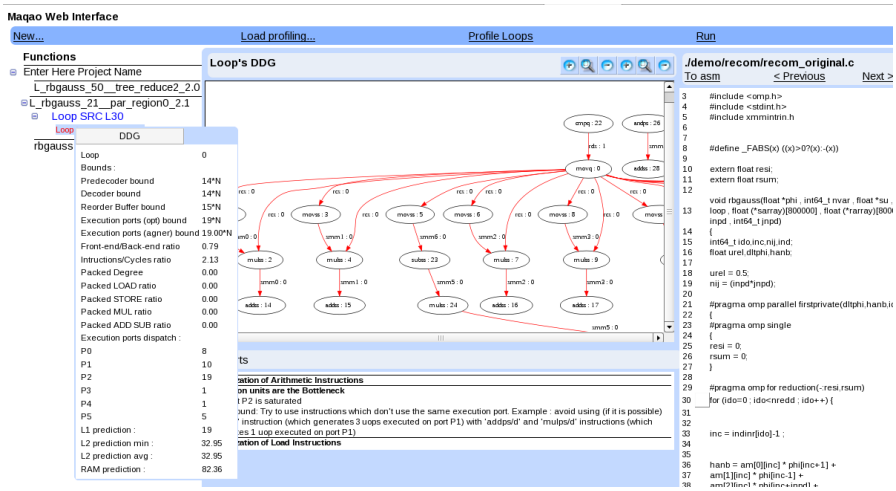

Figure 2. MAQAO interface

2. **Execution Port Usage:** for each of the independent execution ports, MAQAO computes an estimate of the number of cycles spent for one iteration of the loop. This metric is essential to measure the amount of parallelism exploitable between the key functional units: add, multiply, load and store units.
3. **Performance Estimation in L1:** taking into account all of the limitations of the pipeline front end (decoder and permanent register file access limitations, special microcoded instructions) and the pipeline back end, MAQAO provides an estimate of the amount of cycles necessary to execute one loop iteration assuming all operands are in L1. As previously mentioned, this bound is most useful as an optimal lower bound representing peak execution.
4. **Performance Estimations in L2/RAM:** MAQAO computes an estimate for the execution time of a loop iteration, assuming all operands are in L2 or RAM and are accessed with stride 1. This estimation relies on memory access patterns detected at the assembly level and micro benchmarking results on the same memory patterns. The drawback of both of these estimates is twofold: they ignore the stride problem (which in RAM will be essential), and they do not take into account the mixture of hits and misses which is typical of real applications.
5. **Performance Projections for Full Vectorization:** In cases where the code is partially or not vectorized, MAQAO computes performance estimations for 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.
6. **Loop Attribute Profiling:** provides important metrics such as the number of iteration of the loop body and the number of instructions per iteration. This concept will be further developed in Section 2.2.

2.2. Dynamic Analysis - Hardware Counters and Memory Traces

2.2.1. Hardware Counters

Using hardware performance counters can provide us with very useful information on the program execution and, in particular, on how resources are used. Tools such as Intel's

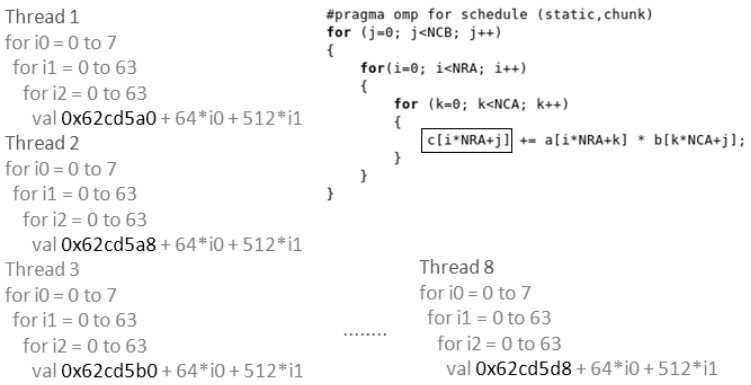

Figure 3. DGEMM 128x128 with 8 Threads and MAQAO trace results

PTU [1], PerfMon [6], PAPI [9], and others make gathering HPM (High Performance Monitoring) information relatively easy. However, even though hundreds of events can be monitored through hardware counters, most of the counters give information that is either too arcane or too esoteric to be useful.

2.2.2. Memory Traces

Memory accesses can dramatically slow down the execution of a program, particularly when it is memory bound. The MAQAO tool is able to provide us with information on the memory behavior of an application. It can track down inefficient memory access behavior in a program by instrumenting instructions dealing with memory. It is also able to catch strides when accesses are done in a regular fashion thanks to the built-in nested loop recognition algorithm. Resulting traces are analyzed to report issues such as false sharing or thread load balancing when using OpenMP. Figure 3 illustrates a typical false sharing issue when misusing the OpenMP schedule type (`for` directive).

If the chunk is sufficiently small - assume we set chunk to 1 - then the schedule will lead to a false sharing issue because threads are all writing on the same cache line. We can see in the traces that the threads' start address (dark hexadecimal values) have only a 1-element distance. Using higher chunk values leads to load issue balancing between threads, only a reduced number of the available threads will be used.

2.3. Decremental Analysis: DECAN

DECAN is an added technique of performance analysis which is used when the static and dynamic analyses are not sufficient to pinpoint the bottleneck. Actually, DECAN is performed manually and consists in: first measuring the original version of the code, and then measuring a version of code modified by removing one or more expressions or instructions. This will of course result in incorrect output of the program, and instructions that will result in a crash or alternate control flow are not removed. Once an instruction is removed, the program is again profiled to account for the contribution of the removed instruction. Timing differences and deltas in L1 and L2 miss rates indicate an individual instruction's effect on a loop's overall performance. DECAN is performed on two levels:

Source Level: here, removing an expression, or more precisely an operand, in an arithmetic expression is simpler and allows a direct correlation between a given source instruction and its impact on performance. However, care has to be taken to make sure that the compiler still performs the same optimizations in both versions.

Assembly level: here the corresponding instruction is replaced by a nop instruction of equivalent size. This case is simpler because we are sure the compiler will not optimize the code differently. However, it is more tedious to reason about the dependences and relate the changes to source instructions.

3. Case study

Here we apply our methodology to a real-life application from Dassault-Aviation.

3.1. Experimental Setup

The experimental platform consists of a computation node equipped with four Xeon X7350 (Tigerton). Each Xeon processor is a quad-core chip cadenced at 2.93 GHz, equipped with two 4 MB L2 caches (two cores share one L2 cache), and 32 kB L1 data cache (private to each core). There are 48 GB of RAM available on this node.

The Intel C and Fortran Compilers (icc and ifort v10.1) are used to generate all our assembly codes and also OpenMP parallel regions when appropriate. Intel's Performance Tuning Utility (PTU) is used to access hardware counters and perform part of the dynamic analysis.

3.2. Iterative Solver for the Navier-Stokes Equation

Brief description The `ITRLSOL` (ITeRative Linear SOLver) [4] application provided by Dassault-Aviation is the linear solver kernel of AeTHER, a larger Computational Fluid Dynamics (CFD) simulation code for the solution of Navier-Stokes equations, discretized on unstructured meshes. `ITRLSOL` is based on an iterative algorithm where the most time-consuming part is located in the `EUFLUXm` subroutine, which implements a sparse matrix-vector product.

`EUFLUXm` contains two groups of 4-level-nested loops (2 identical 4-level-nested loops in each group). The following code snippet displays one of the most time-consuming 4-level-nested loops:

```
         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)
                 enddo
              enddo
           enddo
         enddo
```

Static analysis The MAQAO vectorization report indicates that no vectorization is performed (no use of SSE instructions). The loads cannot be vectorized due to the non unit stride on the two vectors but the multiplications and the additions could have been vectorized by the compiler. However, the Execution Port Usage report clearly indicates that the vectorization of additions and multiplications will improve P0 and P1 ports (execution units ports) but not the P2 port (loads port) which is the bottleneck.

Dynamic analysis Loop attribute profiling allows to detect that the main specific feature in the 4-level-nested loops is that the two innermost loop bounds (ndof) are small ($4 \leq$ ndof ≤ 10). The two outermost trip counts are larger and vary throughout execution.

Memory tracing shows that the two innermost loops are accessing all of the arrays along the wrong dimension (row-wise) leading to poor spatial locality. Moreover, the values of indexes used for accessing the first dimension of all arrays are not regular and lead to indirect addressing.

In the EUFLUXm code, cache behaviour is interesting. PTU allows to detect that for every iteration, a quarter of a cache line, i.e. 64 B/4 = 16 B = 2 double precision elements, are brought into L2. This confirms the fact that the two bi-dimensional vectors are most likely kept into L2 while the two 3-dimensional arrays are streamed from RAM.

Decremental analysis No decremental analysis is performed on Eufluxm. The static and dynamic analyses are significant enough to detect the key performance bottleneck.

Optimization Since the key performance bottleneck for this routine is a poor spatial locality (accesses on the wrong dimension), various transformations are performed. Over the various code transformations that were performed on the code, two have a significant impact on performance: hardwiring ndof and loop interchange.

Value specialization consists in replacing a variable whose value is unknown by the compiler ndof by its proper value (in our case 4) to help the compiler in particular for unrolling. The compiler fully unrolls the two innermost loops inside the loop nest. As no SIMD instructions were generated, MAQAO is fooled by the fact that the innermost loop is not the one it used to be. Hence no direct comparison with the previous reports can be made, except that according to MAQAO no loop vectorization occurred. A speedup of 1.5 is observed for sequential executions.

The second transformation consists in 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. The static analysis of this transformation with MAQAO shows that indirect accesses still prevent the compiler from vectorizing the loop. However, dynamic analysis shows that interchanging loops substantially increases the data traffic into L1 but drastically improves performance. The L2 traffic remains the same but the hardware prefetch behavior is vastly improved. This optimization improves the single core performance by a speedup of 1.5.

In a multicore environment the same optimizations are applied. Variable specialization has an impact on the overall execution of ITRLSOL, but interchanging loops gives even better results, with a speedup of almost up to 2.5, see Fig. 4.

4. Related work

Automatic static analysis of code source generally leads to optimization performed directly inside the compiler [3].

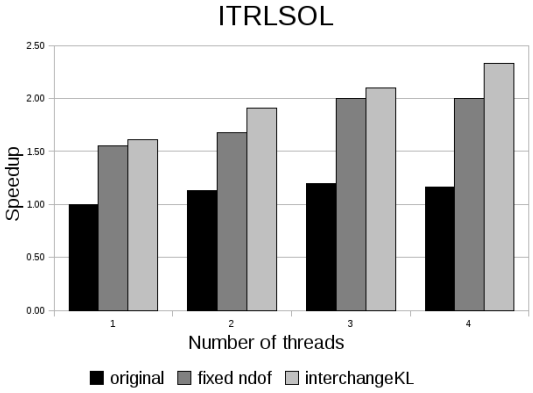

Figure 4. `ITRLSOL` speedups on multicore.

When it comes to performance analysis, dynamic analysis is a natural choice. Efforts have been made to identify the bottlenecks (such as memory contention, communication imbalance leading to idle tasks, etc.) that occur in HPC software [12], as well as a methodology to better understand parallel applications [2]. It introduces a methodology which aims at a better understanding of large-scale HPC applications through the case study of an application which is part of the SPEChpc benchmark suite.

Tallent and Mellor-Crummey propose to measure three metrics to evaluate how well a parallel, multithreaded program performs (parallel idleness, parallel overhead, and logical path profiling) [13]. These criteria then help the programmer to see when to coarsen or refine concurrency granularity, focus on serial performance optimization, or even switch parallelization strategies.

This leads to the design of performance tools dealing with static and dynamic analysis, such as HPCTOOLKIT. It is able to perform binary analysis on an executable (rebuilding loop nests and call graphs, identifying inlined routines) and do a call path profiling at runtime. LoopProf and LoopSampler [8] also work on binary files and focus on loop profiling, with information such as loop properties, nesting, self/total count, trip count, etc. Another framework for instrumentation and measurement of applications is presented by Shande et al. [11]. It describes a suite of performance analysis tool based on PAPI and TAU [10] tools. The profiling with HPM and a tracing tool are applied to extract various types of measurements on Matrix-Multiply and PETSc benchmarks.

5. Conclusion and future work

The methodology presented here provides a semi-automatic way of analyzing and understanding performance issues in the case of High-Performance applications. It combines different tools (MAQAO , hardware counters, value profiling, etc.) used to perform static and dynamic analysis, and possibly decremental analysis for uncovering hidden bottlenecks. Better execution times are achieved in the case of kernels used in real-life applications, with speedups of at least 2.

Nevertheless, there still remains a lot of work to get a better analysis process. Memory traces are analyzed and interpreted by a human being. Detecting the wrong strided accesses due to the wrong loop ordering (such as what was found in the `ITRLSOL` ap-

plication) is automatable to a certain point, and should ultimately be performed by the machine.

Moreover, performing decremental analysis is tedious work best left to a machine. Work is under way to automate DECAN at the binary level. That way, the user will have more control upon what must be suppressed, and what are the consequences, without fearing external intervention (e.g. an optimizing compiler).

Acknowledgments

The research presented in this paper has partially been supported by the French Ministry for Economy, Industry and Employment through the ITEA2 project "ParMA" [17] (June 2007 – May 2010).

References

[1] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization made easier with Intel Performance-Tuning utility, 2007.

[2] B. Armstrong and R. Eigenmann. A methodology for scientific benchmarking with large-scale applications. pages 109–127, 2001.

[3] K. D. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. *SIGPLAN Not.*, 38(2 supplement):97–107, 2003.

[4] Q. V. Dinh, A. NaÃŕm, and G. Petit. Projet fame2: rapport final de synthÃĺse sur l'optimisation des logiciels de simulation numÃl'rique de l'aÃl'ronautique, 2007.

[5] 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, Oct. 2005.

[6] S. Eranian. Perfmon2: a flexible performance monitoring for linux, 2006.

[7] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[8] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 2007 International Conference on Computing Frontiers*, May 2007.

[9] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[10] S. S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.

[11] S. Shende, A. Malony, S. Moore, P. Mucci, and J. Dongarra. Integrated tool capabilities for performance instrumentation and measurement. 2007.

[12] D. Skinner and W. Kramer. Understanding the causes of performance variability in hpc workloads. In *In International Symposium on Workload Characterization*, 2005.

[13] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[14] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 441–452, New York, NY, USA, 2009. ACM.

[15] AMD. Software optimization guide for AMD family 10h processors.

[16] Intel. Intel 64 and IA-32 architectures optimization reference manual.

[17] ParMA: Parallel Programming for Multi-core Architectures - ITEA2 Project (06015). `http://www.parma-itea2.org/`