A Balanced Approach to Application Performance Tuning

Souad Koliai[1], Stéphane Zuckerman[1], Emmanuel Oseret[1], Mickaël Ivascot[1], Tipp Moseley[1], Dinh Quang[2], and William Jalby[1]

[1] University of Versailles Saint-Quentin-en-Yvelines, France
[2] Dassault-Aviation, France
{firstname.lastname}@prism.uvsq.fr,tipp.moseley@colorado.edu,Quang.Dinh@dassault-aviation.com

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, different tools have different strengths. Therefore, an assortment of performance tuning utilities and strategies are necessary to best utilize scarce resources (e.g., bandwidth, functional units, cache).
This paper describes a combined methodology for the optimization process. The strategy combines static assembly analysis using MAQAO with dynamic information from hardware performance monitoring (HPM) and memory traces. We introduce a new technique, decremental analysis (DECAN), to iteratively identify the individual instructions responsible for performance bottlenecks. We present case studies on applications from several independent software vendors (ISVs) on a SMP Xeon Core 2 platform. These strategies help discover problems related to memory access locality and loop unrolling that lead to a sequential performance improvement of a factor of 2.

1 Introduction

In high performance computing, 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. Most often, memory system effects like working set, stride, bandwidth, and alignment are the key factors. Once those are solved, it is important to identify the best instruction scheduling and unrolling factors to keep the pipeline full and balanced. After a particular resource has been identified as a bottleneck, it

is necessary to characterize the specific cause of the problem by pinpointing the specific instructions involved and how they are suboptimal. Using this information, it is finally the responsibility of the programmer to enact a solution.

In practice, this process is extremely tedious because of the difficulty to understand a program's behavior and the complexity of modern processors. As a result, many tools and methodologies exist to analyze a code and guide the optimization. However, the provided results are essentially raw data, requiring an experienced programmer to perform analysis and synthesis of the information. Furthermore, many existing methodologies do not cover the entire process of analysis and perform a subset of the whole analysis (for example, only static analysis).

Hardware performance monitoring (HPM) can provide great insight into a program's performance bottlenecks. However, there are hundreds of performance counters, but only two at a time can be counted on a Core 2. Arcane event descriptions make most performance counters useful only for true micro-architecture experts. Given these problems, we have invested significant time into identifying which performance counters are 1) understandable and 2) correlate well with performance. Using HPM data, we can identify potential bottlenecks and move toward pinpointing a region of code that has potential for optimization.

In this paper, we address optimization of HPC code, specifically CPU- and memory-bound applications. 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 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. Since it uses different and complementary types of analyses (static, dynamic and decremental), our semi-automated methodology is called "balanced".

This methodology has been applied on two industrial HPC codes: `RBgauss` from RECOM Services and `ITRSOL` from Dassault-Aviation. Our modifications to the latter code achieved an improvement in sequential and parallel performance by a factor of 2.

Section 2 presents the tools and techniques in our analysis approach. Section 3 deals with two case studies of HPC codes on which the methodology was applied with significant performance improvement. Section 4 discusses other approaches to performance tuning. Finally, Section 5 concludes.

2 Toward a Better Evaluation Process

The following performance analysis techniques are intended to identify the root cause of a performance bottleneck. It is assumed that the targets of optimiza-

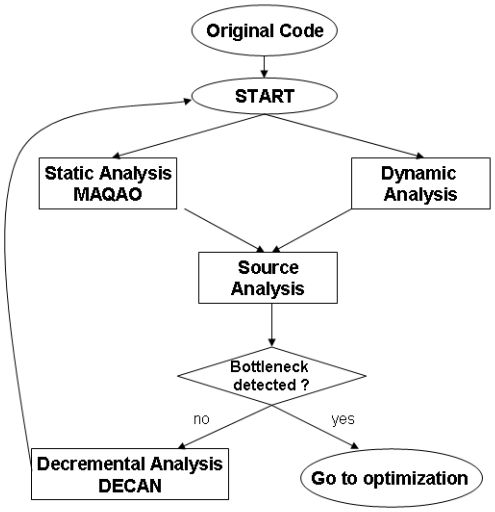


Fig. 1. Methodology diagram.

tion are significant contributors to program execution time as reported by tools `gprof` [9] or Intel PTU [1].

This section provides a high-level description of each step of the methodology. Figure 1 shows a diagram of the flow through the process.

2.1 Static Analysis with MAQAO

MAQAO [6] 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 we have extended it to support x86 programs using an Intel Core 2 model 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:** this report, shown in Figure 3(b) provides individual measures on aligned vector instruction usage (load, store, add, multiply). For example a vector ratio of 1 on the multiply operations means that all of the multiply operations have been vectorized by the compiler. This ratio is computed taking into account only floating point operations and full length vector operations on aligned data. Some compilers report having vectorized loops but without telling how far they go, while others are just silent as to which optimizing transformations are done. For example, ICC reports it has "vectorized" loops involving double precision

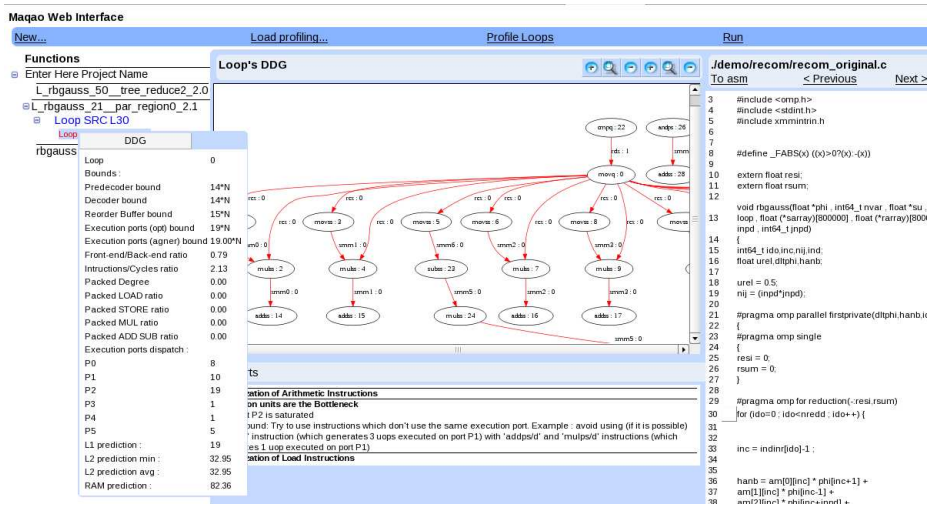
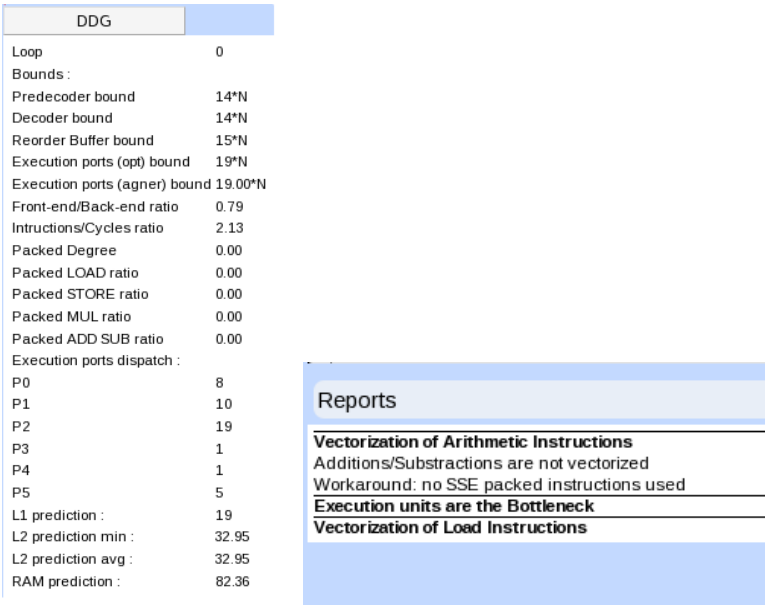


Fig. 2. The MAQAO interface.

values when it generates SSE scalar (64 bits) instructions whereas it should indicate when it generates SSE packed (128 bits) instructions. 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.

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. Figure 3(a) shows the report presented by MAQAO . Since all 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 enables quick detection of whether a code is memory bound and to get a first quantitative estimate of how much it is memory bound.
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 L2 estimate constitutes a reasonable



(a) MAQAO statistics (b) MAQAO reports

Fig. 3. MAQAO interface details.

performance objective while the RAM estimate is a stride 1 worst case. 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. The hit/miss problem can be better analyzed by using decremental performance analysis (Section 2.3) which helps identify the delinquent loads/stores which are accessing the memory.

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 for the x86 ISA. However, in most loops, these indirect access are followed by floating point operations (adds or multiplies) which could be vectorized. The MAQAO performance projection gives a quick estimate of whether trying to vectorize these operations will pay off. The performance projection can also be extremely helpful when combined with the decremental performance analysis. If the latter has determined that all of the load/store operations are accessing operands di-

rectly from memory, there is no point in trying to vectorize memory accesses because they are roughly equivalent in vector or scalar mode.
6. **Loop Attribute Profiling:** MAQAO supports loop attribute profiling which 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.

Source Analysis The correlation made by MAQAO between assembly and source files allows to pinpoint automatically the corresponding source code. For example, if MAQAO notifies some memory accesses (loads) in the execution ports report, the correlation with the source code permits to detect that there are additional loads due to indirect accesses. The source analysis helps to understand the way memory is accessed (directly or indirectly). This stage must be performed manually.

2.2 Dynamic Analysis: Hardware Counters and Memory Traces

Hardware Counters Tools such as Intel's PTU[1], PerfMon[8], PAPI[14], and others make gathering HPM 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. Moreover, only a few events can be monitored at once on most processors. For example, on the Intel Core 2 processors, only 2 different configurable events can be monitored at once in most cases (up to 4 events in very particular cases, such as the monitoring of different variations of a same events). Hence, the first issue to be solved is to identify a limited set of performance counters which should have the following characteristics:

- A small number to avoid numerous reruns which are costly in time.
- Easy to understand and to correlate with performance. On top of the documentation problem, performance counters often refer to low level details of the architecture which are hard to interpret correctly. Understanding the true meaning of many performance counters involves an intimate knowledge of the microarchitecture. Through a painstaking exploration process, we have identified a set of counters that we find to be understandable and correlate well with performance of our target applications. Not surprisingly, the best indicators relate to the memory system. The following hardware counters have been identified to be key indicators of performance for our applications and are understandable.

The INTEL documentation gives the following definitions for the previous counters:

- **L1D_REPL:** Counts the number of lines brought into the L1 data cache.
- **L2_LINES_IN.SELF.ANY/DEMAND/PREFETCH:** Counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the

L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can also count demand requests and L2 hardware prefetch requests together (ANY) or separately (DEMAND/PREFETCH).
- **CPU_CLK_UNHALTED.TOTAL_CYCLES:** Counts the total number of core cycles, while it is running code and while it is halted, as long as it not in a sleep state.

Memory Traces Assembly-level instrumentation is used to collect memory traces to analyze memory access patterns. The memory traces provide a stride report for targeted instructions. Using this stride report, for example, instructions with a longer stride than operand size can be identified as potential performance bottlenecks. An example using this type of information is given in Section 3.2.

2.3 Decremental Analysis

When memory behavior is identified as a problem with a loop, given the imprecise nature of performance counters, it is often still difficult to know the specific delinquent instructions. To quickly identify such instructions, we introduce a technique called decremental analysis (DECAN).

The concept is a simple one: first measure the original version of the code, and then measure a version of code modified by removing one or more expressions or instructions such as memory access 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. For instance, a written variable has to be used after it has been written otherwise the compiler will reduce the code because it will consider the variable useless. 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 2 levels:

1. **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. This can be checked with MAQAO to make sure that the compiler did not further reduce the code or unroll the loops differently.
2. **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 dependencies and relate the changes to source instructions.

Combined with memory tracing (MAQAO), the DECAN approach enables precise identification of the "delinquent" loads with poor memory access patterns. For such loads, a proper optimization strategy will either consist of data

reshaping or adding software prefetch instructions since the hardware prefetcher was clearly not doing a good job on the instruction if DECAN resulted in a noticeable performance gain.

In Section 3.2, DECAN and the memory stride report are used to identify delinquent loads. We have found there is a significant benefit in understanding by approaching performance bottlenecks from multiple angles, especially when they arrive at the same conclusions.

3 Case Studies

Here we show how the methodology previously described was applied to real-life applications. In the remainder of this section, we will examine two code excerpts, developed by Recom Services and 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 clocked 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, as they are state-of-the-art compilers on such a hardware platform. They also are used to generate 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 Application to a 3D Combustion Simulation Code

Brief description The `AIOLOS`[15] application provided by RECOM builds a 3D model of industrial-scale furnaces, and in particular, helps solve problems due to the corrosion of the walls of such a furnace at high temperatures. The most time-consuming subroutine in `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 two loops consists of :

- Red loop: it is an iterating loop over half of the `AM` array elements (*red* elements) to update with the other half of the `AM` array elements (*black*). It means that each *red* element depends on its four immediate *black* neighbors.
- Black loop: it has the same structure as the red loop but it updates the black elements with the red ones (computed in the Red loop).

The following code snippet displays one of the two time-consuming loops previously described:

```
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 = UREL*( HANB/AM(INC,7) - PHI(INC) )
 PHI(INC) = PHI(INC) + DLTPHI

 RESI = RESI + ABS(DLTPHI)
 RSUM = RSUM + ABS(PHI(INC))
ENDDO
```

Static analysis According to MAQAO , no vector instructions (SSE instructions) are generated by ifort. However, looking at the Execution Port Usage report, it becomes apparent that the main bottleneck is memory accesses, specifically loads from memory. The `P2` port (memory loads) shows a much higher score (i.e. the number of accesses to `P2` per iteration) than the other ports.

Looking at the source code, the explanation for not using vector instructions in the loop is obvious: the `AM` array is accessed indirectly through the `INDINR` index array, which prevents the compiler from knowing whether data accessed are correctly aligned, even when told so with compilation directives.

Dynamic analysis As both Red and Black loops are singly-nested loops, the iteration count and the bound are one and the same for each loop. Memory tracing indicates that the `AM` and `PHI` arrays are accessed with a stride 2 basis, with some gaps from time to time. As the static analysis shows, the most time consuming operations in this loop are memory loads. Looking at the source code and loop attribute profiling, it is clear that this routine is memory-bound. As `AM` is accessed with a "stride 2" pattern, half of the memory bandwidth is wasted: only half of the bytes pulled into the cache are actually useful for performing computations, doubling the number of cache misses. Looking at performance counters confirms this. Overall, a basis for future optimization has been issued.

In a multicore context, dynamic analysis shows a constant amount of L1D and L2 cache line consumption, independent of the number of threads used. Thus, the amount of cache misses per thread is constant, but the amount of CPU cycles increases, thus reducing the overall speedup to 4 with 16 threads. This is mainly due to memory bandwidth limitations which prevent memory bound programs from getting more than a speedup of four.

Decremental Analysis Removing expressions (at the source level) one by one has an impact on the analysis given by MAQAO by decreasing load pressure on port P2. However, since MAQAO cannot distinguish between different strides, its added-value remains limited in this particular case. However, when iterating the dynamic analysis with the modified loop, memory behaviors become more apparent: accesses to `PHI` occur almost always in cache, whereas accesses to `AM` are always RAM-based. The dynamic analysis is applied after the decremental

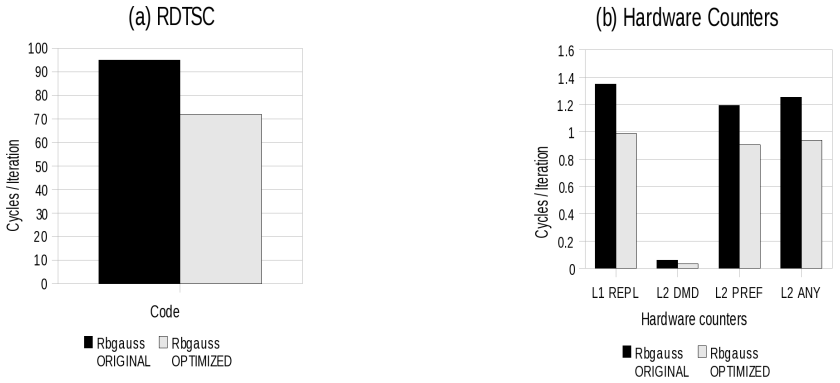


Fig. 4. `RBgauss` code optimization on unicore.

analysis to account the effect of the removed instruction. DECAN was essential in this case to identify which memory accesses are causing the contention on the memory bus. Even if the first application of DECAN does not detect the delinquent instruction, the analysis is reiterated (removing other instructions) combining with static and dynamic analyses. In the `RBgauss` subroutine, the blocking memory acces is detected with DECAN.

Optimization 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.

As expected, the indirect access still prevents the compiler from generating vector instructions. As such, MAQAO is still "blind" to our code transformation. However, dynamic analysis, and more specifically hardware counter measurements do show that cache misses are almost half what they used to be (Figure 4(b)). Single core performance has been improved by speedups between 1.2 and 1.3 (Figure 4(a)) thanks to this code transformation.

When performing a new dynamic analysis with multiple threads, memory saturation is still the main problem, but saturation of the memory bus happens much later. Hence performance is improved by speedups between 1.3 and 1.4.

3.3 Iterative Solver for the Navier-Stokes Equation

Brief description The `ITRSOL`[5] application, developed by Dassault-aviation, solves the Navier-Stokes equation, through the use of Computational Fluid Dy-

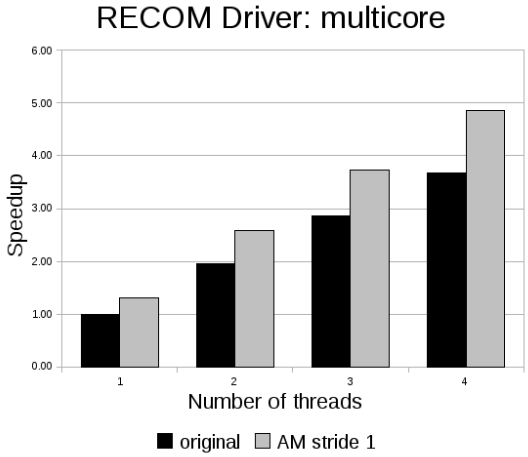


Fig. 5. `RBgauss` speedups on multicore.

namics (CFD), with the help of an iterative solver. The most time-consuming subroutine in `ITRSOL` 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).

The following code snippet displays one of the most time-consuming quadruply 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.

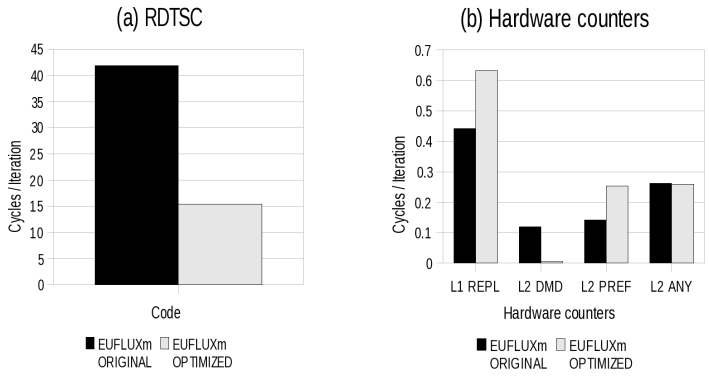


Fig. 6. `EUFLUXm` code optimization on unicore.

Dynamic analysis Loop attribute profiling indicates that the main specific feature in the 4-level-nested-loops is that the two innermost loop bounds ($ndof$) are quite small ($4 \leq ndof \leq 10$). 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 behavior is interesting. PTU detects that for every iteration, a quarter of a cache line (16 bytes, 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 Decremental analysis is not necessary for the `Eufluxm` benchmark. After applying optimizations deduced from the static and dynamic analyses, the key performance bottlenecks are mitigated.

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

Value specialization involves 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.

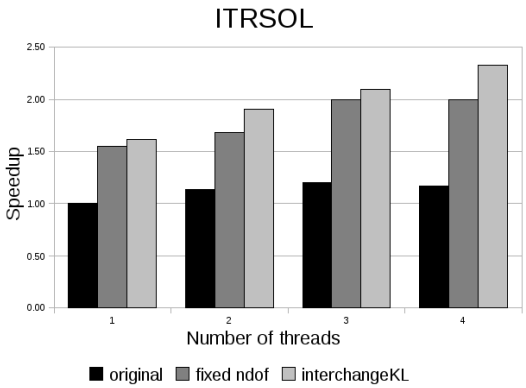


Fig. 7. `ITRSOL` speedups on multicore.

The second 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. The static analysis of this transformation with MAQAO shows that indirect accesses still prevent the compiler from vectorizing the loop. Statically, there is no information about the transformation. However, dynamic analysis shows that interchanging loops substantially increased the data traffic into L1 but drastically improved performance.Because of the size of the data set, the L2 traffic remains the same, but the hardware prefetch behavior is vastly improved Figure 6(b). Figure 6(a)) shows this optimization improves sequential performance by a speedup of 2.5.

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

There is no doubt that improving data locality in a unicore environment has similar effects on a multicore environment, as the memory bus receives fewer requests. As previously discussed, speedup for memory-bound applications on the experimental platform cannot be greater than 4 due to bus saturation. Since the previous experiment was conducted on a 4-thread execution and gave a speedup around 3.5, it was decided to keep the same amount of threads for `ITRSOL`.

4 Related Work

Automatic static analysis of code source generally leads to optimization performed directly inside the compiler [4]. Dolan et al. present a formula to measure (and then compare) the sensibility of solvers (optimization software) to input data [7].

Hochstein et al. introduce a methodology to reduce the development time of an HPC application in addition to its execution time [10]. 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 [18], as well as a methodology to better understand parallel applications [3]. It introduces a methodology which aims a better understanding of large-scale HPC applications through the case study of an application which is part of the SPEChpc benchmark suite.

Other efforts have been made to go further, such as using data mining techniques to exploit a knowledge database and apply transformations on the code to optimize [20]. PerfExplorer [11] also uses data mining to find interactions between groups of performance metrics and to tune applications for large clusters.

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) [19]. 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 [13] 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. [16]. It describes a suite of performance analysis tool based on PAPI and TAU[17] tools. The profiling with HPM and a tracing tool are applied to extract various types of measurements on Matrix-Multiply and PETSc benchmarks.

Finally, numerous processor vendors propose guides to apply efficient optimization techniques to reduce execution time [12, 2]. However, they generally tend to be extremely low-level (at the assembly level) and do not really describe how to diagnose a bottleneck.

5 Conclusion and Future Work

While the tools and methodology presented in this paper represent progress toward making the optimization process easier and less time consuming, much work still remains in streamlining the analysis process. The reports gathered via memory tracing are currently processed manually, so there are opportunities automatic analysis to identify the key components. Decremental analysis shows great potential, but it is currently subject to trial and error. We are developing tools to automatically identify dependent instructions to safely patch instructions at the binary level without having to consider potential interactions with the compiler.

In this work, we present a methodology to provide a semi-automatic way of analyzing and understanding performance issues for high-performance computing applications. This was done using a combination of different tools: MAQAO , HPM counters, loop attribute profiling, and memory tracing are used to perform static and dynamic analysis. Decremental analysis is used to identify the root cause of certain bottlenecks. Better execution times are achieved for kernels used in real-life applications, with speedups of up to 2.5.

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. AMD. Software optimization guide for amd family 10h processors.
3. B. Armstrong and R. Eigenmann. A methodology for scientific benchmarking with large-scale applications. pages 109–127, 2001.
4. K. D. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. *SIGPLAN Not.*, 38(2 supplement):97–107, 2003.
5. Q. V. Dinh, A. Nam, and G. Petit. Projet fame2: rapport final de synthse sur l'optimisation des logiciels de simulation numrique de l'aronautique, 2007.
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, Oct. 2005.
7. E. D. Dolan and J. J. Mor. Benchmarking optimization software with performance profiles, 2001.
8. S. Eranian. Perfmon2: a flexible performance monitoring for linux, 2006.
9. 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.
10. L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 35+, Washington, DC, USA, 2005. IEEE Computer Society.
11. K. A. Huck and A. D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 41, Washington, DC, USA, 2005. IEEE Computer Society.
12. Intel. Intel 64 and ia-32 architectures optimization reference manual.
13. 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.
14. 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.
15. B. Risio, N. Passmann, F. Wessel, and E. Reinartz. 3d-flame modelling in power plant applications. 2008.
16. S. Shende, A. Malony, S. Moore, P. Mucci, and J. Dongarra. Integrated tool capabilities for performance instrumentation and measurement. 2007.
17. S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.
18. D. Skinner and W. Kramer. Understanding the causes of performance variability in hpc workloads. In *In International Symposium on Workload Characterization*, 2005.
19. 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.
20. V. S. Verykios, E. N. Houstis, and J. R. Rice. A knowledge discovery methodology for the performance evaluation of scientific software. *Neural, Parallel & Scientific Computations*, 8:115–132, 2000.