The Design and Architecture of MAQAOAdvisor: A Live Tuning Guide

Lamia Djoudi, Jose Noudohouenou, and William Jalby

Université de Versailles, France
lamia.djoudi@prism.uvsq.fr

Abstract. Program performance is tightly linked to the assembly code, this is even more emphasized on EPIC architectures. Assessing precisely quality of compiled code is essential to deliver high performance. The most important step is to build a comprehensive summary for end-user and extract manageable information. In this paper, we present our first prototype called MAQAOAdvisor, a key MAQAO(Modular Assembly Quality Optimizer) module that drives the optimization process through assembly code analysis and performance evaluation. It performs comprehensive profiling, hot-loop and hot-spot detection, fast evaluation and guides local optimizations. An originality of *MAQAOAdvisor* is to deport part of optimizations from the driver to a post-compiler evaluation stage. It is based on static analysis and dynamic profile of assembly code. It feeds information back to help end-user detect and understand performance problems. It proposes optimization recommendations to guide a user to perform the best transformations to get the best performance.

1 Introduction

The quest for performance leads to an ever increasing processor complexity. Similarly compilers are following the same trend with deeper optimization chains involving numerous sets of techniques. As a result code performance is becoming more and more complex to guarantee, it is sensitive to butterfly effects and difficult to assess without extensive tuning and experiments. The Three fundamental points for code optimization are to detect, understand and fix potential performance problems. Nowadays this issue is mostly tackled by using hardware counters and dynamic profiling. An array of tools is used to handle these three stages of performance tuning. Consequently, tuning is a time consuming task, burdensome with a poor productivity. Therefore, a modern approach is much needed, to address the complexity of the task in order to support the multidimensional aspect of performance and complemented existing methods.

We propose an approach which allows us to find the best orientation to guide a user to perform the best transformations to get the best performance. Understanding of how and why the compiler bottleneck occurs, through the feed-back of more information, helps us to execute the code much faster.

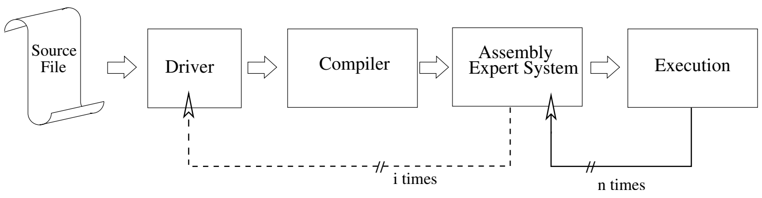


Fig. 1. Adding a stage is a way to cut through the cost of evaluation (by preventing useless execution) as well as to limit the number of evaluations (by preventing the iterative process to apply useless optimizations)

The novelty of our approach is:

1-The optimization part: Which is transferred from the driver to a post-compiler evaluation stage. Being after the compilation phase allows us a precise diagnostic of compiler optimization successes and/or failures, or if due to some obscure compiler decision, the resulting code contains under-performing patterns. Assembly level is the natural place to observe performance, because it is close enough to the hardware and it is possible to check the job done by the compiler. The idea is to enrich the performance ecosystem with a new actor in a collaborative way with the compiler.

2-The assembly code analysis: Our approach gives the first decision about the code quality and which transformation should be applied to improve the quality of assembly code and by consequence it's performance. The use of both static analysis and dynamic profiling within a single framework seems to provide a great amount of flexibility for designers to try out new optimization patterns. By combining static and dynamic analysis, we centralize all low level performance and build correlations.

3-The modification of iterative compilation process: As depicted in Figure 1, our system includes an extra stage between the compiler and the execution. Our approach is located between a model-driven optimization and with machine learning optimization without training. The advantage of this method is to have less N executions than the original iterative compilation so, we speed-up the execution time of the search engine. The driver keeps track of the different transformations to apply next. It reads a list of transformations that it needs to examine together with the range of their parameters. With the original approach, we have only the feedback with the execution time or a few hardware counters. In our approach, we can have more detailed information on the assembly code with an expert system which is in charge of collecting information from an inner-view perspective in contrast with execution time or hardware counters which provide an outer-view. Furthermore, the feed-back provided to the compiler is richer than simple raw cycle counts. This feed-back contains the set of pre-selected transformations than the expert system supposed to be relevant.

In this paper, we present our first prototype called MAQAOAdvisor which is used to provide a live tuning guide capable of improving performance and/or code quality that is not caught by existing tools. It aims to simplify the understanding of the compiler optimizations. To answer the question: is it possible to learn a decision rule that select the parameters involved in loop (application) optimization efficiency ?. We build a summary that defines an abstract representation of loops(application) in order to capture the parameters influencing performance.

MAQAOAdvisor advocates a new approach which can be combined with traditional iterative compilation. This module is characterized by a finer granularity and a richer feed-back. It alleviates the cost of iterative compilation and enlarges the spectrum of candidate codes for optimization.

MAQAOAdvisor, a key MAQAO module drives the optimization process through assembly code analysis and performance evaluation. It is fully implemented in MAQAO (information is presented to the user in a hierarchical manner in a GUI application) MAQAO[1] is a tool which allows the analysis, the manipulation and the optimization of assembly code generated by the compiler. MAQAO tries to identify the optimizations done (or not) by the compiler. Developing *MAQAOAdvisor* as an expert system seems to be a suitable answer as the other generic methods that are not adapted to the highly specific problem of code optimization. It implements a set of rules to help end-user to detect and understand performance problems and make optimization recommendations to guide a user to perform the best transformations to get the best performance.

This paper is organized as follows: Section 2 details MAQAOAdvisor overall design. Section 3 illustrates MAQAOAdvisor outputs. Section 4 details the guided optimization. Section 5 presents related work. And we conclude in Section 6.

2 Overall Design of MAQAOAdvisor

Gathering data and statistics is necessary for a performance tool, but it remains only a preliminary stage. The most important step is to build a comprehensive summary for end-user and extract manageable information. *MAQAOAdvisor* acts as an expert system to drive user attention within the performance landscape. Providing an expert system to help the user to deal with complex architecture was done by CRAY's AutoTasking Expert [2]. It was focused on parallelization issue and was neither as extensible nor as sophisticated as MAQAO's performance module. *MAQAOAdvisor* is built over a set of rules and metrics:

2.1 Performance Rules

Relying on static as well as dynamic information, MAQAOAdvisor implements a set of rules to help end-user to detect and understand performance problem. All rules are written with the support of MAQAO API which allows manipulating MAQAO internal program representation and quickly writing compact rules. Rules can be sorted in three categories:

Transformations Rules: Once assembly code parsing data are stored, the application of the transformations rules will format and gather them according to some conditions in a data table. We detailed four transformations rules:

Issue cost per iteration, jointly with cycle cost, this metric allows to evaluate the cost of data dependences for the loop. A large gap induced by data dependency hints that the loop should be unrolled more aggressively or targeted by other techniques to increase the available parallelism.

Cycle cost per iteration, is expressed as a function of the number of iterations, for non-pipelined loop it is simply in the form of: $a \times N$ where N is the number of iterations. This static cycle evaluation is a reference point to estimate the effectiveness of dynamic performance.

Theoretical cycle bounds per iteration, estimate the data dependency weight in the critical path. This metric[3] indicates if the loop is computationally or memory-wise bound. Knowing whether a loop is computationally or memory-wise bound is a powerful indicator of the kind of optimization techniques to use. Typically computationally bound loops imply that lots of cycles are available to tolerate memory latency problems.

Pipeline loop, where the cost function is: $a \times N + b$. N being the number of iterations, a the cost per iteration and b the filling-up/draining pipeline cost.

Deduction Rules: From the data table of transformation rules, other rules are deduced to help the end-user to (1) detect and understand the performance problem, (2) search effective optimizations, (3) understand optimization failures and obscure compiler decision and to propose code transformations. The deduction rules can be sorted in three categories:

High Level Rules, add semantic to assembly code loop structures. Based on heuristic they are able to compute unrolling factor, degree of versioning, inlining, presence of tail code and report suspicious pipeline depth. These rules also evaluate cost of data dependencies, compute the gap with bound of optimality or hint for vectorization opportunities. Some rules are also dedicated to estimate the purpose of loop versioning. The main cost of loop versioning is the introduction of (a limited) decision tree overhead to select the relevant version, and code size expansion. Several optimizations bring an improvement large enough to overcome this additional cost, but when the gain is questionable, versioning should be turned off.

Code Pattern Rules, are dedicated to rules based on known bad code patterns. For instance on Itanium 2, in some cases the couple of `setf/getf` instructions are used to convert values from the general purpose to the floating point register file. These conversions are costly and in some cases available. Therefore it is valuable to report presence of such patterns. Additionally some rules based on pattern matching evaluate if loops are performing `memcpy` or `memset`. With MAQAOPROFILE, we can have the number of iterations. Also, in MAQAO, we have a summary about some specific functions. For example, the insert of `memcpy` is interesting when the number of iterations is greater than 1000. In this case, a message is reported advising to modify the source code and insert a library call. Additionally `spill/fill` operations are detected, as well as memory operations prone to bank conflicts. On the source level, MAQAOAdvisor also detects if a code is badly written and proposes some high level transformations.

Low Level Rules, address performance problems due to some architecture specifications. For instance in Itanium architecture, it can be: *branch buffer saturation* with 1 cycle long loop body (i.e. one branch to process every cycle). Hardware can not sustains the branch throughput and this leads to some extra stall cycles of the pipeline. Other architecture specific problems like register pressure, or lack of prefetch instruction in a loop with memory operations, and so on.

Additional Rules: MAQAOAdvisor is a library of high level rules which can be extended according to user needs. Users can easily extend the MAQAOAdvisor by writing their own rules.

2.2 Hierarchical Reporting Approach

The needs of the end-user differ, depending on which *level*, the decision is going to be made: is it to chose between two compilers? To select different compilation flags for the whole application? To tune specifically a given loop? Being aware of this, MAQAOAdvisor organizes information hierarchically. Each level of the hierarchy is suitable for a given level of decision to be taken: complete loop characterization, loop performance analysis, function or whole code analysis.

At the first level, the instructions are coalesced per family (e.g. integer arithmetics, load instructions) and counted on a per basic block basis.

The second level, which is already an abstraction layer, only reports loops where some important performance features are detected, thus filters out a large amount of non-essential data. Additionally results are reported in a user-friendly way. This level summarizes the tables of:

(i) selected instruction counts and built-in metrics are displayed which require some knowledge to be interpreted but they represent the exact and complete input of what MAQAO is going to process in the upper stages. However the goal is to detail instructions that have been determined as being of special interest.

(ii) instruction count enriched by built-in metrics : *Cycle cost per iteration*, *issue cost per iteration* and *theoretical cycle bounds per iteration.* Together counts and metrics are exploited by *MAQAOAdvisor rules* which process results gathered during application execution (instrumentation, hardware counters, cycle counts).

(iii) versioning summary for each hot loop. The idea is to perform a study of different versions based on the number of iterations, to decide which is the best version for each interval of iterations, to classify the versions as function of the number of iterations, and choose for each interval of iterations the best one in order to improve parallelism in the original code or in the new optimized code (very interesting to improve the compositional versioning[4]).

The third and forth level, respectively, for each routine and the whole code, a report counting the number of detected performance issues. Reading these tables is quick and was designed to facilitate comparison.

The fifth level summarizes different optimizations. When *MAQAOAdvisor* orients user to generate different versions of each hot loop, MAQAO has the possibility to perform a global study (static analysis, profiling) for all versions at the same time. This automatic process is the "mode project" in MAQAO.

The sixth level, gives a comparison between different transformations, i.e. for the same code, compiled with different compiler flags; it is possible to do a paired comparison (graphical or tabular). One can perform this comparison for each level (1 to 4) or generate a comparison report.

The seventh level, performs code comparison, i.e. for the same code, compiled with different compilers; here also it is possible to do a paired comparison.

3 MAQAOAdvisor Outputs

Based on the static and dynamic results at all levels, MAQAOAdvisor sorts functions, loops and projects by their respective weight.

3.1 MAQAOAdvisor Modes

MAQAOAdvisor results are displayed in the MAQAO interface or in a report by using the batch mode. MAQAOAdvisor rules and those written by users, can be applied automatically to a large set of files in batch or interactive mode.

3.2 Static Analysis Results

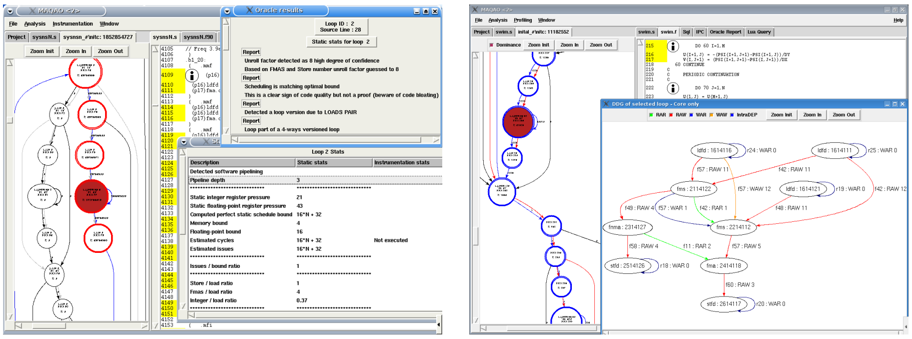
As an analyzer, MAQAO's static module extracts the entire code structure. The structure is expressed through a set of graphs. These graphs are simple yet powerful to analyze a code. Several types of static analysis are also displayed in MAQAOAdvisor. It provides a diagnosis of selected functions, loops or basic blocks like the number of instructions and the information about inner loops.

Call Graph (CG): By selecting one function in CG, MAQAOAdvisor gives all its loops static/dynamic information.

Control Flow Graph (CFG): Represents the predecessor/successor relation among basic blocks and facilitates to display MAQAOAdvisor results for one selected loop (see Figure 2 (a)).

Data Dependency Graph (DDG): Computing the DDG is a key issue to (1) determine critical path latency in a basic block, (2) perform instructions re-scheduling or any code manipulation technique, (3) allow an accurate understanding of dynamic performance hazards, (4) determine the shortest dependency that corresponds to the overlapping bottleneck (see Figure 2 (b)).

Versioning: If the user chooses one loop and click on *versioning button*, MAQAO provides a new window with a summary of the versions of this loop generated by the compiler. If he had performed an instrumentation before, and he clicks on *graph versioning*, *MAQAOAdvisor* provides the distribution of loop iteration count for each version. This information helps us to decide which optimization and version is the best. At this level, *MAQAOAdvisor* can also give a guiding report to do better optimization.



Loop ID : 2
Source Line : 28

Static stats for loop 2

Report
Unroll factor detected as 8 high degree of confidence
Based on FMAS and Store number unroll factor guessed to 8

Report
Scheduling is matching optimal bound
This is a clear sign of code quality but not a proof (beware of code bloating)

Report
Detected a loop version due to LOADS PAIR

Report
Loop part of a 4-ways versioned loop

Loop 2 Stats

Description	Static stats	Instrumentation stats
Detected software pipelining		
Pipeline depth	3	
******************************	******************************	
Static integer register pressure	21	
Static floating-point register pressure	43	
Computed perfect static schedule bound	16*N + 32	
Memory bound	4	
Floating-point bound	16	
Estimated cycles	16*N + 32	Not executed
Estimated issues	16*N + 32	
******************************	******************************	
Issues / bound ratio	1	
******************************	******************************	
Store / load ratio	1	
Fmas / load ratio	4	
Integer / load ratio	0.37	
******************************	******************************	

(a) MAQAOAdvisor displays analysis. (b) Data Dependence graph

Fig. 2. SPECFP 2000 benchmark. (a) 178.galgel: close inspection of the loop loop b1_20. In front of each loop of the source code, ⓘ gives access to the information computed by the MAQAOAdvisor concerning this loop. (b) 187.facerec: DDG of hot loop in gaborRoutine. User can choose RAW, RAW, WAR, WAW or intra dependence. It can also visualize them at the same time.

Static Statistics: are the representation of transformations rules detailed in section 2.1 and they can be displayed in the MAQAO interface.

3.3 Dynamic Analysis Results

MAQAO proceeds to code instrumentation automatically[5]. It measures the real application behavior with minimal disturbance. An interesting side effect of our instrumentation is its very low run-time overhead. Profiling information is used to build an execution summary, they can be transparently accessed by end-user or used by MAQAOAdvisor.

3.4 Combining Static/Dynamic Analysis

Example of static/dynamic results:

Prefetch impact: By applying prefetch transformation rules, MAQAOAdvisor detects if a loop containing load or store instructions does not contain prefetch. It warns and advocates for first checking the source code (to consider if data streams are manipulated) and if necessary to use prefetch intrinsics. Intrinsics force the compiler to generate prefetch instructions. This prefetch warning is not emitted in the case of loop tail code, because loop tail codes have only a limited number of iterations. In such a case, the lack of prefetch instruction makes sense.

Value Profiling Results: Time profiling is of limited help for such a fine granularity, but value profiling leads to numerous optimizations. For instance, it is the key metric for code specialization. Additionally, extracting some characteristics of address streams is useful to prevent bank conflicts, aliasing problems or to detect the prefetch distances. Prefetch distances could theoretically be computed off-line with an assembly code analysis. However, it remains easier and safer to rely on dynamic traces, since for instance on Itanium architecture some optimizations allow a single prefetch instruction to retrieve several data streams.

Summary Analysis: By comparing static and dynamic analysis, MAQAOAdvisor detects the value undecidable by a pure static scheme and gives more information to take the best decision.

4 Guided Optimization

MAQAOAdvisor helps end-user to navigate through his code and isolate the particularly important or suspicious pieces of code. For these isolated pieces which are the hot loops, *MAQAOAdvisor* provides as many guidances as possible to help the decision making process. This "guided-profile" allows to understand the compiler optimizations and guides to improve code quality and performance. As detailed in Figure 3, *MAQAOAdvisor* is designed as a set of interlinked levels each of them being loosely coupled to the others. User can take decisions at the end of each level. The best decision is taken at the end of the process.

4.1 Automatic Hot Loops Selection

In this stage, we must find the hot loops to be optimized. MAQAOPROFILE[5] allows us to give a precise weight to all executed loops, therefore underscoring

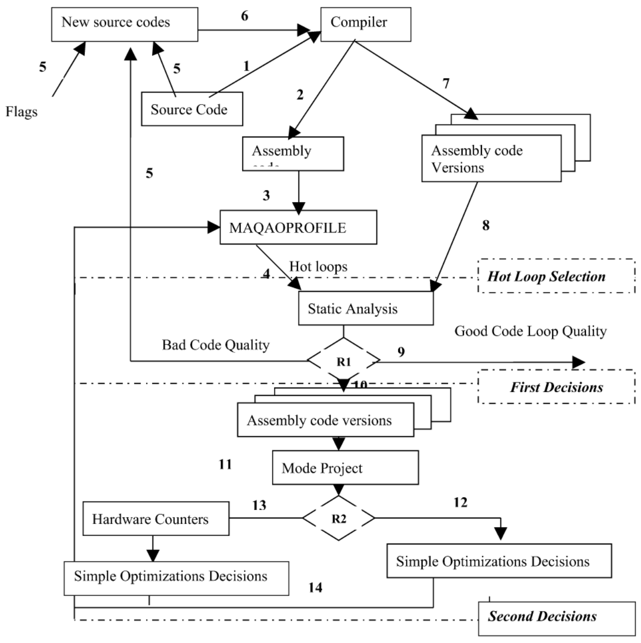


Fig. 3. MAQAOAdvisor Process

hotspots. Correlating this information provides the relevant metrics: (1) Identifying the hotpath at run-time which passes through the whole program where the application spends the most of its time is a key for understanding application behavior. (2) Monitoring trip count is very rewarding, by default most of compiler optimizations target asymptotic performance. Knowing that a loop is subjected to a limited number of iterations allows us to choose the optimizations that characterized by a cold-start cost.

4.2 First Decisions

Based on static analysis, *MAQAOAdvisor* takes first deductions of compiler optimizations and proposes to:

(i) Add "pragma" to avoid (1) the register pressure in order to avoid the spill/fill, (2) the check instructions (that mean compiler had take a bad optimizations), (3) and to inform the user that lot of calls can decrease performance.
(ii) Improve code quality in order to improve performance. Code quality depends of the first ratio $R1$ (issue/bound). It evaluates the matching between static bounds[3] and observed performance. If it is equal to one, then the function/loop is removed from the list of optimization candidates. Otherwise, candidates are evaluated according to several factors: value profiling is used to detect stability.

If we have just one version with $R1 \leq 1.2$ and there is no problem of spill /fill, check instructions and functions calls, *MAQAOAdvisor* decides that is the best one and the process can be stopped here.

If $R1 \geq 1.3$, we generate the first guided optimization. It combines the static and dynamic analysis of the original version of each hot loop. Then it allows user to apply the first optimization for the hot loops in source or assembly level to improve code quality and the performance. For example, it can propose optimization on source level, like software pipelining, unrolling, add prefetch.

At this level, applying different transformations for several hot loops in assembly or source level, implies the generation of several versions of code. The analysis of these versions allows to find the best version or what kind of transformations user must take, to have the best performance at the second level of the *MAQAOAdvisor*. It is possible that the compiler may not improve the code quality, so *MAQAOAdvisor* orients user to the second decisions.

4.3 Second Decisions

Once at this level, we are sure that we can improve the performance more than the previous level. To find the trade-off between quality and performance it is interesting to calculate the second ratio $R2$.

$R2 = \frac{\mathrm{c}_2(N)}{\mathrm{c}_1(N)}$ where: $\mathrm{c}_2(N)$ = number of cycles executed for N iterations.

$\mathrm{c}_1(N) = A_1.N + B_1$, where A_1 is the static cycles of the body, B_1 is static cycles spent in overheads and N is the number of iterations of the loop.

If the compiler unrolls the original loop and generates a remainder loop, the formula of c_1 is: $\mathrm{c}_1(N) = A_1.N + B_1 + a_1.(N \bmod UF) + b_1$ where a_1 and b_1 are the parameters of the loop corresponding to remainder iterations and $(N \bmod UF)$ is the remainder iterations and UF is the unrolling factor. This ratio answers the question: Does the static code represent a good dynamic behavior?

To take a decision to what we do, *MAQAOAdvisor* combines the information like $R1$ and $R2$ values of one or more versions for each hot loop:

Simple Optimizations decisions: *MAQAOAdvisor* follows this path for the good $R2$ value ($R2 \leq 3$) and decides to guide user to:

Combining best versions in the same source code, where $R1 \leq 1.2$. It is a high level optimization. To achieve a trade-off between code quality and performance, *MAQAOAdvisor* combines for each hot loop and their best versions: (1) the unrolling factors, (2) the loop and code size, (3) $R1$ and $R2$. All this process is automatic. This information is given to a solver that finds the trade-off. *Rescheduling*, where $R1 \geq 1.2$. A generation of the DDG of the loop can help us to reschedule the assembly instructions in order to improve the code quality. We choose the version that corresponds to the small $R2$.

Compositional loop specialization, we can also apply a low level optimization. It's indepent of the $R2$ value and it can complete and give more performance than the two first optimizations. Knowing the number of iterations, this technique[4] can generate and combine sequentially several versions at the assembly level. We can get best performance with this technique because we improve the best versions using the MAQAOAdvisor decisions.

Complex Optimizations decisions: If we have a bad $R2$ value ($R2 \geq 3$), *MAQAOAdvisor* guides user to use hardware counters. An interesting advantage, the hardware counters are implemented in MAQAO. Executing a simple script in MAQAO, *MAQAOAdvisor* combines the hardware counters and MAQAO results to guide user to take a decision. For example to solve the cache misses, *MAQAOAdvisor* can propose one of the decisions:

Memory reuse, by modifying the stride of the loop or aggregating the data.

Optimization cache, taking a copy of data or a blocking cache decrease the TLB.

Recovery of Data access latencies, by adding a pragma in source code or modify the prefetch distance in assembly code. This modification is still in progress in the compositional approach implemented in MAQAO.

4.4 Optimization Results

In this section, we evaluate our proposed technique. We consider three benchmarks: CX3D application, a JACOBI code, and a benchmark from the SPECFP2000.

Experiments were run on a BULL Itanium 2 Novascale system, 1.6GHz, 3MB of L3. On the software side, codes were compiled using Intel ICC/IFORT 9.1.

CX3D: CX3D is an application used to simulate Czochralski crystal growth a method applied in the silicon-wafer production. It covers the convection processes occurring in a rotating cylindrical crucible filled with liquid melt.

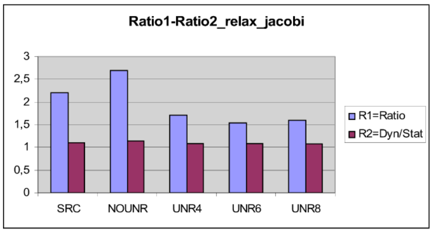
Based on hardware counters technique, we count the cycles, instructions and nop retired as well as back_end_bubble_all stall cycles, we remark that back_end _bubble_all stall cycles are the most important (more than 50%). To know the reason of this stall, we must analyze the subevents. The BE_L1D_FPU_BUBBLE dominates (86.21%). To know the reason of this stall, we must observe two events where BE_L1D_FPU_BUBBLE_L1D takes 99.54%. To have more precision, we observe different sub-events for this event. The cause in this level is that the compiler had a problem to load integer variables from L1D in one cycles (BE_L1D_FPU_BUBBLE_L1D_DCURCIR takes 51%). Arriving at this level, we do not have more precise information and we must take another approach to understand the problem.

But if we use our process, we are sure that we take less time than trying to understand the hardware counters results in order to identify the problem and then give a solution. With our approach, firstly we can just base on static analysis giving the first diagnostic. Combining static and dynamic analysis, our system can give a precise diagnostic and a precise solution to improve performance. For example, for this loop, one of the suggestions is the memory access aliasing. Our aliasing memory module proves that we have an aliasing problem. After that a precise solution proposed by our system is "you must apply an interchange". The process is organized as follows: first a fine grain profiling is done to get accurate hot functions and for the hot functions we give the accurate hot spots. Then the most time consuming inner loops are optimized according to their static and dynamic analyses of our method.

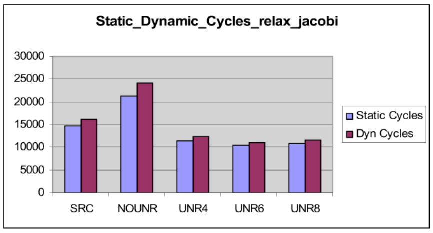
1 - MAQAOPROFILE Information:

(i) Hot functions: the time attributed to the highest routine (velo) is 70.12 %.
(ii) Hot loops: to isolate the most time consuming loops. The hot loop which is at the source line 787. Other loops have been omitted for sake of clarity.

2 - Optimization sequence: Based on MAQAOAdvisor process, we try to improve code quality in order to improve performance. Our approach is applied to loop id 75. Before applying different transformations and relying only on the combination of static and dynamic information, MAQAOAdvisor (i) collects compiler optimization information applied to this loop , (ii) proposes different solutions (unrolling, prefetching and interchange) for this loop. Generating these versions and the summary of the static information, the GLPK solver indicates that the interchange transformation is better. The gain is 60% and R1 becomes 1 (good code quality). To prove our approach, we have also applied profiling for these transformations. We have remarked that there is a correspondence between solver solution or proposition and dynamic results. That proves, it is not necessary to execute different versions in order to find the best execution time corresponding to the best transformations. But just with the useful static information, we can find the best transformations.

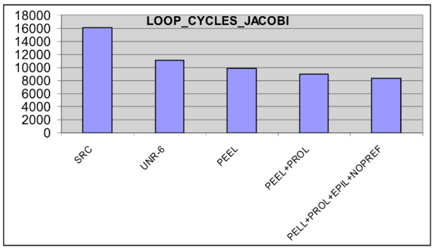


(a) Static guided-profile

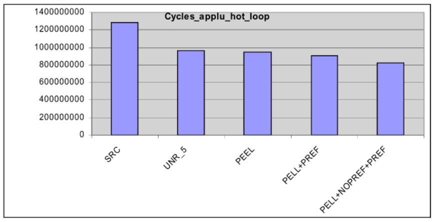


(b) Dynamic guided-profile

Fig. 4. Hot file (relax_jacobi), hot loop (2655 source line), selected versions: (a) first and second ratio. (b) static cycles and dynamic cycles.



(a) Compositional versioning of jacobi



(b) Compositional versioning of Applu

Fig. 5. CPU cycles for different compositional versioning: (a) loop 23 (source line) in relax_jacobi, (b) loop 2655 (source line) of 173.applu

JACOBI: Jacobi code solves the Helmholtz equation on a regular mesh, using an iterative Jacobi method with over-relaxation. The first level of our approach demonstrates that Jacobi contains one important hot loop (source line 2655). This level allows us to generate some versions of this loop using pragma. Introducing all guided-profile important information to the GLPK[10] solver, it finds a trade-off and decides the version unroll 6 is the best one (see Figure 4). For the best version, we have applied different transformations. See Figure 5(a).

173.APPLU: It is a benchmark from SPECFP2000 which leads to the performance evaluation of the solver for five coupled parabolic/elliptic partial differential equations. The same process of jacobi was applied for this benchmark and the best version is the version unroll 6. Accurate results of compositional versioning are provided in Figure 5(b).

5 Related Work

Very few tools focus at providing user with transformation code advices for performance tuning. Tools such as foresys [6] or FORGExplorer [7] propose code analyses as well as code transformations but no techniques to identify the tuning transformation to use. Vtune[12] is mainly a profiling tool. Its usage is so widespread that an API gets standardized to describe their access. CAHT[8] shares the same goal as Vtune: "discover performance-improvement opportunities often not considered by a compiler, either due to its conservative approach or because it is not up-to-date with the latest processor technology". CAHT also formalized the search of tuning advices and so builds an easily extensible system based on case-based reasoning (CBR). The solution proposed by CAHT is not precise when there are no similar cases because it must ignore some characteristics to provide a solution. We propose to extend the MAQAOAdvisor to incorporate the case-based reasoning but just for similar case. With each new case and the use of an expert system, we are sure we will enrich the knowledge base with very precise cases. In addition to the combination between the CBR and expert system, we will propose precise solutions.

`ATOM` [11] and `Pin` [15] instrument assembly codes (or even binary for `Pin`) in a way that when specific instructions are executed, they are caught and user defined instrumentation routines are executed. While being very useful `Atom` and `Pin` are more oriented toward prospective architecture simulation than code performance analysis. EEL[9] belongs to the same categories of tools. This C++ library allows editing a binary and adding code fragment on edges of disassembled application CFG. Therefore it can be used as a foundation for an analysis tool but does not provide performance analysis by itself. Currently EEL is available on SPARC processors. Vista [13], is an interesting cross-over between compiler and performance tool. Plugged with its own compiler, Vista allows to interactively test and configure compilation phases for code fragment. Everything is done in a very visual way. While being conceptually close to MAQAO, Vista remains more a compiler project than a performance analyzer.

Shark [14] offers a comprehensive interface for performance problems. Like MAQAO, it is located at the assembly level for its analyzes, displays source code as well as profiling information. As most of Apple's software the GUI is extremely well designed. However Shark lacks instrumentation and value profiling. Code structures are not displayed and the Performance Oracle advices are currently limited to very few messages: alignment, unrolling or altivec (vectorization). Additionally as most of Apple's software it is very proprietary and does not offer open-source scripting language or standard database. Nevertheless it remains an advanced interface, with an extensive support of dynamic behavior and it underlines the need to think performance software beyond gprof.

6 Conclusion

MAQAO is a tool that centralizes performance information and merges them within a representation of the assembly code. MAQAO also provides several views on the internal representation of an input program that the user can navigate through. MAQAOAdvisor module drives the optimization process by providing support through assembly code analysis and performance evaluation. It explores the possibility to let a user interact with program analysis and opens new ways of exploring, modifying and optimizing assembly and source code.

Taking advantage of precise profiling information, our system is able to select the most suitable optimizations among a list defined by the user (Deep Jam, ...) or using directives (unroll, software pipelining, prefetch,...). A trustable API to drive exactly the sequence of optimizations through the compiler would be useful to unleash the potential of our (any) feed-back approach.

Our goal is to improve MAQAOAdvisor to a real expert system. The idea is to present a prototype which is a design expert system for MAQAO incorporating case-based reasoning. The case-based reasoning is the method in which we create a knowledge base. If you have a new application, the system searches a domain dependent case-base for a similar case:

(i) If there is one, the system uses it to propose solutions to improve performance and/or code quality with minimum user interaction. To do this we must analyze the application and identify its characteristics and its context.
(ii) When there is no similar case, instead ignoring certain characteristics (the case of case-based reasoning), we can leave it to the user or an intelligent system. With each new case and the use of an expert system, we are sure we will enrich the knowledge base with very precise cases. In addition to the combination between the CBR and expert system, we will propose precise solutions.

As a compiler construction tool, our framework can be useful to compare different compilers. For instance, it is easy to track regressions between two versions of a compiler or to have an accurate picture of compilation flag impact.

The main goal is to export MAQAO to different VLIW compilers. The implementation of Trimedia architecture is still in progress. The assembly code generated by Trimedia is very similar to Itanium 2 and we implement an interface in MAQAO in order avoid writing another MAQAO specific to Trimedia.

In the future, MAQAO will include improving optimization module by adding new optimization techniques, based on powerful mathematical models.

References

1. Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., Jalby, W.: MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In: Workshop on EPIC, San Jose (2005)
2. Kohn, J., Williams, W.: ATExpert. Journal of Parallel and Distributed Computing 18(2), 205–222 (1993)
3. Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., Jalby, W.: Exploring Application Performance: a New Tool for a Static/Dynamic Approach. In: Los Alamos Computer Science Institute Symposium, Santa Fe, NM (2005)
4. Djoudi, L., Acquaviva, J.-T., Barthou, D.: Compositional Approach applied to Loop Specialization. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 268–279. Springer, Heidelberg (2007)
5. Djoudi, L., Barthou, D., Tomaz, O., Charif-Rubial, A., Acquaviva, J.-T., Jalby, W.: The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module. In: Workshop on EPIC, San Jose (2007)
6. FORESYS, FORtran Engineering SYStem, `http://www.pallas.de/pages/foresys.htm`

7. FORGExplorer, http://www.apri.com/
8. Monsifrot, A., Bodin, F.: Computer aided hand tuning (CAHT): applying case-based reasoning to performance tuning. In: ICS 2001: Proceedings of the 15th international conference on Supercomputing (2001)
9. Larus, J.R., Schnaar, E.: EEL: Machine-Independent Executable Editing. In: The ACM SIGPLAN PLDI Conference (appeared, June 1995)
10. http://www.gnu.org/software/glpk
11. Srivastava, A., Eustace, A.: ATOM - A System for Building Customized Program Analysis Tools. In: PLDI 1994, pp. 196–205 (1994)
12. VTune Performance Analyzer, http://www.intel.com/software/products/vtune
13. Zhao, W., Cai, B., Whalley, D., Bailey, M., van Engelen, R., Yuan, X., Hiser, J., Davidson, J., Gallivan, K., Jones, D.: Vista: a system for interactive code improvement. In: Proceedings of the joint conference on Languages, compilers and tools for embeded systems, pp. 155–164 (2002)
14. http://developer.apple.com/tools/shark_optimize.html
15. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation Micro 37, Portland (2004)