Submitted for confidential review to: *The 2007 International Symposium on Code Generation and Optimization*

The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module

Abstract

This paper presents MAQAOPROFILE, an instrumentation MAQAO module. MAQAOPROFILE builds a comprehensive profile fine grain profile of the application. In order to build a more accurate profiling, instrumentation have to be handled after the compilation stage. This is the only way to prevent compiler optimizations to stumble on extra-code added for the purpose of profiling. The instrumentation module gives MAQAO the possibility to execute rules based on both static and dynamic analysis. Consequently, instrumentation is performed by injecting a limited number of extra-instruction, named *assembly probes*, around the targeted code fragments to monitor. This low-level scheme is technically challenging to implement (e.g. ensuring the integrity of the register stack for IA64 code) but it allows minimal interferences with both the behavior and performance of instrumented applications. In fact, the instrumentation effects have to be as reduced as possible not to perturbate original code behavior.

Instrumentation can focus at several levels of granularity, each one having its own interest. Additionally to timing, instrumentation is also performing *value profiling*. Value profiling provides an observation of the application from an inner point of view, at the opposite of the traditional profile which remains solely behavioral. As MAQAO is able to perform analysis on multi sources (project mode), MAQAOPROFILE instruments those project. Implemented in MAQAO tool, MAQAOPROFILE provides several views on the control flow graph, call graph and analysis module. These views can be used to navigate through the assembly or source code, to edit analysis results in different manner, and a performance advisor is implemented to help end-user to detect and to understand performance problem. It indicates optimizations compiler, fails optimization and its solutions.

1. Introduction

The evolution of the technology and architectures of the processors, get always a maximum theoretical performances. However, the really performances obtained by the programs are far from the ideal performances. The degradation causes can be from: the source code, operating system, hardware architecture or compiler.

In order to cure these sources of degradations, efforts under development research are mobilized to bring tools and solutions to understand, to predict and to evaluate the code performance. The goal of these works is above to reduce this variation of the performances.

Computer architects need tools to evaluate how programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing or to provide input for profile-driven optimizations. Program analysis tools are extremely important for understanding program behavior.

Most of the performance analysis tools/toolkits can be dispatched among two main classes. The first one is focused on the exploitation of hardware performance counters while the second relies on code instrumentation or even transformation. The regular use of performance instrumentation and analysis tools to tune real application is surprisingly uncommon. traditionally, hardware counters, profiling information, static analysis and even expert knowledge are exploited individually or at best interconnected through ad hoc tools.

As a response to the enlarging gap between needs and existing software, tool we have developed MAQAO, which stands for Modular Assembly Quality Analyzer and Optimizer. The concept is to centralize all low level performance information and build correlations. As a result, MAQAO produces more and better results than the sum of the existing individual methods. MAQAO is designed as a set of interlinked modules each of them being loosely coupled to the others.

MAQAO [1] is working at assembly level: Assembly code is a gold mine of information. These information just need correlation and a way to be exploited (either brought back to end user and fed directly in the compilation chain) to reveal their high value. Additionally, being based after the compilation phase allows a precise diagnostic of compiler optimization successes and/or failures. Thus reporting directly to end-user information on potential improvement.

As an automatic tool, MAQAO processes large amount of data and applies optimization and diagnostic to the whole code and is not restricted to a limited number of hot spots. This approach allows to track down most of the little percentage loss all over the code.

Finally, we rely on existing standard solutions when they are effective: hardware counters are supported through perfmon [2], data storage is handled with a database (which can queried by SQL). And a scripting language is embedded within MAQAO to allow user to extend it according to his own needs.

Our purpose is not to design yet another software tool but to implement an optimization methodology and to propose a real Performance Framework. Assembly code inspection is done statically, data profiling is done using code instrumentation and hardware counters fit their traditional role of hotspot detection.

MAQAO 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 (including call stack, garbage collection, binary analysis), and it underlines the need to think performance software beyond gprof.

This paper presents MAQAOPROFILE, an instrumentation module in MAQAO tool. A complement module to MAQAOPROFILE is MAQAORACLE. It is a performance advisor driving the optimization process through assembly code analysis and performance evaluation. The rest of this paper is organized as follows. In section 2, we outline the MAQAO framework. We describe briefly in section 3 the static analysis. Section 4 provides the overall design of MAQAOPROFILE. Section 5 describe the architecture and implementation of MAQAOPROFILE. In section 6, we describe brievely MAQAORACLE. In section 7, we point the reader to related work in the area of instrumentation tools. Finally, in section 8, we state our conclusion.

1.1 Motivating Example: Hardware Counters Are Not The Panacea

Performance is often a multi-dimensional problem, this is obviously due to the number of actors involved and their respective complexity.

Despite the multi-faceted nature of performance analysis, the current trend is to rely heavily on hardware counters. Results obtained by this approach are numerous and of high quality, therefore counters are now ubiquitous on every processor hitting the market. Additionally counters are more or less standardized mainly due to the PAPI [3] initiative. However, while being helpful this uni-dimensional method is not sufficient and often answers only to a part of the performance question. For instance it is well known that what is important *is not the number of cache misses, but if these misses were overlapped or not*. In fact, a large amount of the hardware budget is already spent for latency tolerance techniques (more than 96.3% of the 1.72 Billion transistors on Montecito processor [4] if caches are considered as belonging to such techniques), therefore in term of efficiency these features should be used, but performance trouble appears when they are overused.

Obviously the miss cost problem could be tracked down with a thorough performance counter analysis coupled with a large set of experiments: however the cost to pay for that shows that counters are not the adequate tool. The following example, coming from a real world optimization problem, emphasizes hardware counters power and limitation.

1.1.1 Experimental Set up

All the experimental results reported in this paper were obtained with a state of the art compiler, *icc* v9.0. Experiments were run on a 1.6 GHz / 9 MB L3 Itanium 2 system.

Loop trip	2	4	8	16	32	64
CPU Cycle	87	150	278	548	1065	2101
Inst. issued	304	542	1018	1970	3874	7682
F. Ops issued	68	136	272	544	1088	2176
Stall Cycle	18	36	72	157	307	607
Stall %	21%	24%	26%	29%	29%	29%

Table 1. Hardware counter measurements for FFTW 4 codelet. Stall cycle is fairly limited, and IPC seems satisfying. Overall, performance is around 33 CPU cycles per iteration.

Loop trip	2	4	8	16	32	64
FP. instr.	68	136	272	544	1088	2176
MEM instr.	44	88	176	352	704	1408
Bound (cycles)	28	56	112	224	448	896

Table 2. Corresponding Essential Instructions determined by MAQAO static analysis. Code appears to be compute bound with an optimal execution time of 14 cycles per cycles.

1.1.2 Optimizing FFTW kernels

FFTW (Fastest Fourier Transform of the West) [5] is a highly tuned version of the FFT. FFTW is built over decomposition of complex FFT in simple computational blocks named *Codelets*.

These codelets, which are small pieces of code, contain most of the computation. A generator is able to produce automatically different flavors of codelets but all of the codelets are generated in C, thus performance depends on the compiler. Trying to push performance beyond the best found compiler options (`-O3 -fno-alias`), starts with a close examination of hardware counters.

Essential instructions: determining intrinsic code bounds

Results from Table 1 depict dynamic results, i.e. a code with good IPC and a rather limited fraction of stall cycles. Dynamic and static results match for the number of instructions issued: 304 instructions are issued in two iterations, 542 for four iterations, corresponding to an average of 120 instructions issued per iteration plus 64 instructions of overhead (the same applies for cpu cycles). The dynamic measurement is around 33 cycles per iterations.

Stalls are determined statically as the difference between the instructions issues and the cycles prediction provided by the compiler. In our case (simple code without branch), stalls are stemming mostly from dependencies (load to use latencies, floating point latencies ...). The compiler predicts 5 stall cycles per iteration while the hardware counter measured 9 stall cycles per iteration. Adding this 4 extra cycles to the number of cycles estimated by the compiler, gives 32 cycles per iteration which is very close to the 33 cycles measured. Therefore, optimizing the code by reducing the number of stalls would bring at best a 20% speed-up.

The performance issue is somewhere else: MAQAO reports key information on loop structure: loop is neither pipelined nor unrolled. MAQAO also collects static metrics on the assembly code and lists the following essential instructions:

34 Floating point operations [1]:

$$\left.\begin{array}{ll} 6\ \text{FMA} & -2\ \text{per cycle} \\ 22\ \text{simple flops} & -2\ \text{per cycle} \end{array}\right\}\ 14\ \text{cycles}$$

22 Memory operations:

$$\left.\begin{array}{ll} 14\ \text{loads} & -4\ \text{per cycle} \\ 8\ \text{stores} & -2\ \text{per cycle} \end{array}\right\}\ 8\ \text{cycles}$$

If we consider a perfect overlap between memory and computational operations (data dependencies completely concealed), bound is $\max(memory, floating\ point)$. Therefore this loop is compute bound: 14 cycle per iteration. Table 2 allows a quick comparison with static analysis results.

While dynamic measurements return a cost of 33 cycles per iteration, the optimal static schedule is 14 cycles. This exposes the code bloating problem: among all the issued instructions, how many are *essential/useful* instructions ?

Monitoring dynamically the stream of addresses manipulated by the loop reveals that addresses appear in sequential order. This means that iterations are not interleaved nor overlapped. Therefore performance is either constrainted by strong data dependencies or parallelism is weakly exploited. Simple analysis of the source code indicates that the iterations are independent. However, the source code contains read and write array accesses of the form $A(1)$, $A(ios)$. Not knowing that ios is strictly positive forces the compiler to have a very conservative schedule. Following these deductions, using the classic versioning (or specialization) optimization for the given loop trip allows to provide the compiler with the critical information that ios value is never set to 0. This time, the compiler generates a much more efficient code, performance comparisons are provided in figure 1, where the optimized version delivers a speed-up of 40% as soon as iterations are over 8. To sum up, a tool assessing assembly code quality is essential to produce high-performance code. Indeed, even state-of-the-art compilers do generate poor quality assembly from real codes and this is difficult to evaluate using a pure dynamic approach.

[1] One FMA counts as 2 floating point operations.

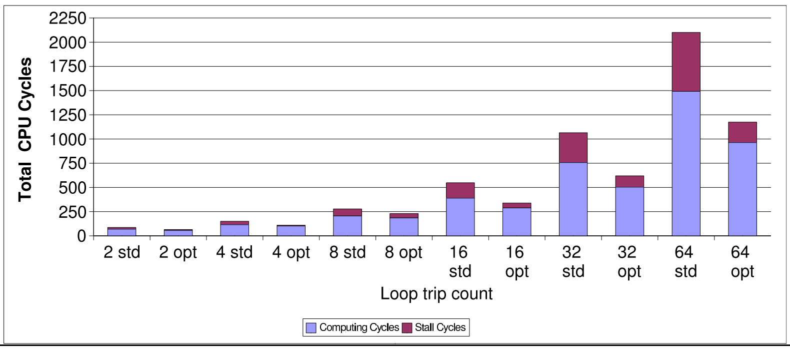


Figure 1. FFTW4 Codelet Performance. CPU cycles are displayed for the standard (*std*) and (*Opt*) which is the version after code specialization. Performance is down from 33 cycles per iteration to 18 cycles per iteration. The fundamental aspect is that the observed improvement is greater than the number of originally measured stall cycles (in red). Thanks to non essential code elimination. Hardware counters are blindly reporting computing cycles (in blue) while the CPU is processing useless instructions.

A stage of static analysis delivers interesting results, and is a shortcut to optimizations w.r.t. to the long and burdensome hardware counters analysis.

2. The MAQAO Framework

For all our experiments and analyzes, we resort to MAQAO. MAQAO combines static and dynamic analysis of assembly codes, on the Itanium platform, in order to exhibit any shortcomings in the compiler optimization process and correlate dynamic behavior with static analysis in order to improve code efficiency. The module of importance for the methods presented in this paper are:

Static analysis: Takes the assembly code as input, parses it and extracts code structure in the form of control flow graph, call graph and data dependence graph.

Dynamic analysis: MAQAO injects assembly probes at various parts of the code. At execution time these probes record time stamp and also data manipulated by the code. Profiling information is used to build an execution summary. The probes can be accessed by end-user or used by the Oracle module.

Performance advisor: MAQAO is an interactive tool that helps users to navigate through the program code and isolate particularly important pieces of the code. It provides a thorough guidance to help the decision making process. However, higher level of decision can be taken by the end user, such as selecting specific compilation flags or splitting the code and compile different parts with different flags. At this point the interactive mode of MAQAO is not very helpful and does allow the correct level of code quality observation.

Optimization: The optimization can be done by modifying the instruction code, the data layout or any other element that is involved in the execution of a program. Our optimization methods are based on: (1)modifying assembly code,(2)rescheduling assembly code (e.g. peeling), (3)merging different versions (e.g. preftch/noprefetch).

Modules are centered around a core module and a database which is ultimately the place where every piece of information is stored. Database is in charge of ensuring data persistence and also offers a standardized storage format.

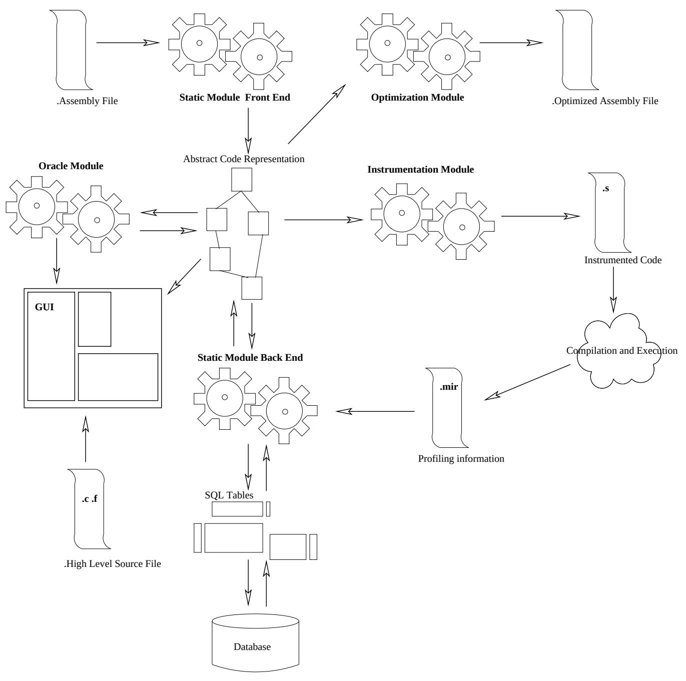


Figure 2. MAQAO components and module organization. The backbone is the abstract code representation, a set of data structure federalizing all program performance information. This information is manipulated by all modules. To ensure data persistence they are dumped in a database.

The MAQAO framework supports the following features. First, it allows a user to view a low-level graphical representation of the code. Second, it analyzes the assembly code generated by compiler(IA32, IA64, x86, gcc). Third, it can give to the user all necessary information (e.g. pipelined loop, exploitation of the resources). Forth, a user can instrument the assembly code. Fifth, MAQAO provides an expert system to help user to deal with complex architecture and to understand (1)optimization failures, (2)obscure compiler decision and (3)propose effective optimizations. As detailed in figure 2, Data are displayed in a GUI to offer a way for end-users to navigate among all the amount of code information. Still to leverage end-user effort, MAQAO provides a LUA interface to access most its internal data structures. A user can write, based on this API, his own requests which fit to their particular problems. In this section, we describe the functinary of MAQAO.

3. Static Analysis Module

Close inspection of assembly code is real mine of information. MAQAO allows to do it in a systematic and structured way. As a matter of fact, manipulating and understanding flat code is difficult. The lack of relief and hierarchy implies an important effort to filter out essential pieces from low interest part of the code. As an analyzer MAQAO's static module extracts code structure. The structure is expressed through a set of graphs: Call Graph (CG), Control Flow Graph (CFG) and Data Dependencies Graph (DDG). These graphs are simple yet powerful tools to analyze a code.

Notice that for MAQAO a fundamental granularity is the innermost loop. Innermost loops constitute critical code fragment since they are the source of large fraction of the execution time. Most of MAQAO's analyzes target this code granularity, making of MAQAO a software well suited for scientific applications.

This notion of innermost loop is expressed in Itanium assembly by the usage of counted branch for the back-edge (and not branch driven by compare instructions).

- *Issues cost per iteration*, this metric is an overoptimistic estimation of the schedule. Basically it is reporting only the number of cycles needed to emit all loop's instructions. Therefore, only resources are taken into consideration. Jointly with cycles cost, this metric allows to evaluate the cost of data dependences for the loop. A large gap induced by data dependencies hints that the loop should be unrolled more aggressively or targeted by other techniques to increase the available parallelism (loop fusion, hoisting and so on).

- *Cycle cost per iteration*, directly extracted from comments let by *icc* is the code. If needed can be trivially computed on the base of processor's specification [6], [7]. The cost is expressed as a function of the number of iterations, whereas for non-pipelined loop it is simply in the form of: $a \times N$. N being the number of iterations. This static cycle evaluation is an the reference point to estimate the effectiveness of dynamic performance.

- *Theoretical cycle bound per iteration*, estimates the data dependency weight in the critical path. This metric indicates if loop is of nature compute / memory bound [8]. As introduced in section 1.1 by the motivating example, at some point it is important to know the ratio of important instructions compared to 'syntactic sugar' code. A well written code should exhibit a bound close to the issues cost and close to the cycle count. Knowing whether a loop is compute or memory bound is a powerful indicator of the kind of optimization techniques to use. Typically compute bound loops implies that lots of cycles are available to tolerate memory latency problem.

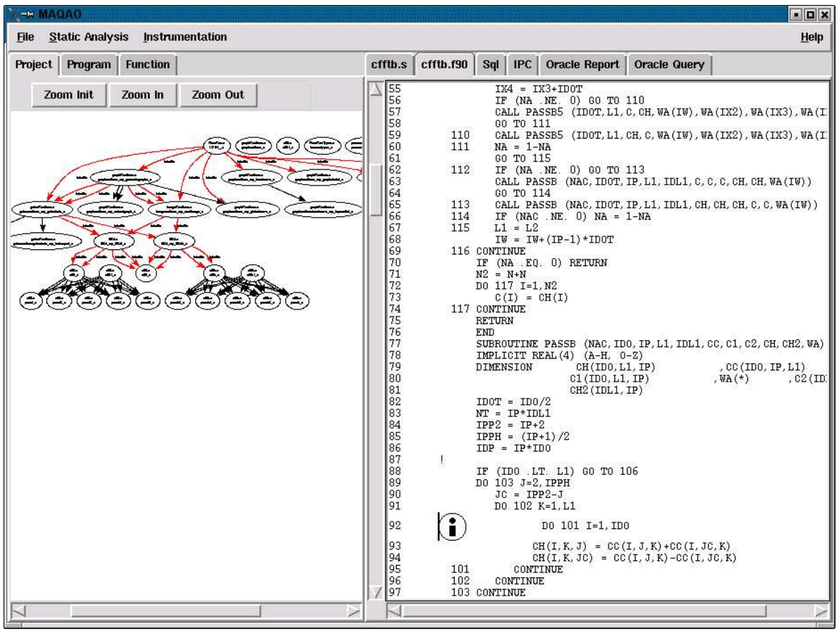
- *Pipeline depth*, warming up and draining a deep pipeline is costly, and this can be overpriced in case of limited number of iterations. A pipeline loop has a different cost function than a regular loop. Since to achieve at least the equivalent of the first full iteration of the source code, a pipeline loop needs to execute as many iterations as the pipeline contains stages. Thus, there is a warming up cost for pipelined loops (which should be paid off by a better throughput when the number of iterations increases). 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.

MAQAO computes several metrics which should trigger attention. Relevance of these simple statistics was illustrated in section 1.1.2. Among the leading metrics:

4. Overall Design of MAQAOPROFILE

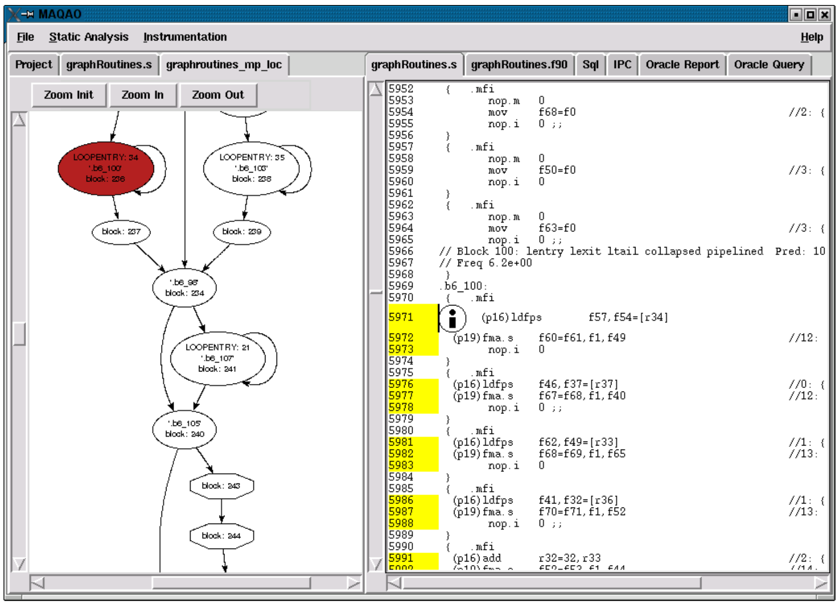
MAQAOPROFILE consists of four componenents, as depicted in figure 5.

At first MAQAO takes as input assembly files generated by compiler (IA32, IA64, x86) and parses them to produce a structured representation of the assembly code. This representation is explained in section ...



```
                IX4 = IX3+IDOT
                IF (NA .NE. 0) GO TO 110
                CALL PASSB5 (IDOT,L1,C,CH,WA(IW),WA(IX2),WA(IX3),WA(I
                GO TO 111
      110       CALL PASSB5 (IDOT,L1,CH,C,WA(IW),WA(IX2),WA(IX3),WA(I
      111       NA = 1-NA
                GO TO 115
      112       IF (NA .NE. 0) GO TO 113
                CALL PASSB (NAC,IDOT,IP,L1,IDL1,C,C,C,CH,CH,WA(IW))
                GO TO 114
      113       CALL PASSB (NAC,IDOT,IP,L1,IDL1,CH,CH,CH,C,C,WA(IW))
      114       IF (NAC .NE. 0) NA = 1-NA
      115       L1 = L2
                IW = IW+(IP-1)*IDOT
      116    CONTINUE
             IF (NA .EQ. 0) RETURN
             N2 = N+N
             DO 117 I=1,N2
                C(I) = CH(I)
      117    CONTINUE
             RETURN
             END
             SUBROUTINE PASSB (NAC,IDO,IP,L1,IDL1,CC,C1,C2,CH,CH2,WA)
             IMPLICIT REAL(4) (A-H, O-Z)
             DIMENSION       CH(IDO,L1,IP)          ,CC(IDO,IP,L1)
                            C1(IDO,L1,IP)          ,WA(*)       ,C2(ID
                            CH2(IDL1,IP)
             IDOT = IDO/2
             NT = IP*IDL1
             IPP2 = IP+2
             IPPH = (IP+1)/2
             IDP = IP*IDO
   !
             IF (IDO .LT. L1) GO TO 106
             DO 103 J=2,IPPH
                JC = IPP2-J
                DO 102 K=1,L1
                       DO 101 I=1,IDO
                      CH(I,K,J) = CC(I,J,K)+CC(I,JC,K)
                      CH(I,K,JC) = CC(I,J,K)-CC(I,JC,K)
      101          CONTINUE
      102       CONTINUE
      103    CONTINUE
```

Figure 3. SPEC FP 2000 benchmark 193.Facerec scrutinized with MAQAO. On the left pane, the static call graph is computed. Red arrows represent call across different source files, while black arrows function calls within the same source file. Here all the files used to compile the benchmark are displayed, the amount of Red arrows is a straightforward metric to estimate inlining opportunities. Notice on the right tab, where the source code of one file is displayed, a ⓘ which indicates that on a single mouse click MAQAO displays high level analysis for this loop.



```
    {   .mfi
          nop.m    0
          mov      f68=f0                                       //2: {
          nop.i    0 ;;
    }
    {   .mfi
          nop.m    0
          mov      f50=f0                                       //3: {
          nop.i    0
    }
    {   .mfi
          nop.m    0
          mov      f63=f0                                       //3: {
          nop.i    0 ;;
   // Block 100: lentry lexit ltail collapsed pipelined  Pred: 10
   // Freq 6.2e+00
    }
   .b6_100:
    {   .mfi
            (p16)ldfps       f57,f54=[r34]
     (p19)fma.s    f60=f61,f1,f49                               //12:
          nop.i    0
    }
    {   .mfi
     (p16)ldfps    f46,f37=[r37]                                //0: {
     (p19)fma.s    f67=f68,f1,f40                               //12:
          nop.i    0 ;;
    }
    {   .mfi
     (p16)ldfps    f62,f49=[r33]                                //1: {
     (p19)fma.s    f68=f69,f1,f65                               //13:
          nop.i    0
    }
    {   .mfi
     (p16)ldfps    f41,f32=[r36]                                //1: {
     (p19)fma.s    f70=f71,f1,f52                               //13:
          nop.i    0 ;;
    }
    {   .mfi
     (p16)add      r32=32,r33                                   //2: {
```

Figure 4. SPEC FP 2000 benchmark 193.Facerec: close inspection of function `graphroutine_localmove` with MAQAO. Part on the CFG is displayed on the left pane. Loops appear clearly with their back-edge. Hexagonal blocks (at the bottom) corresponds to basic blocks including function call. The assembly code matching the selected loop (in red) is highlight in yellow on the right pane. ⓘ indicates that on a single mouse click MAQAO displays low level analysis on this loop detailed in figure 8.

MAQAO depends on an instrumentation file to guide the instrumentation process and on a static analysis file to specify the code that must be executed at run-time.

- *Instrumenter:* injects a limited number of extra-instruction, named *assembly probes*, around the targeted code fragments to monitor. This low-level scheme is technically challenging to implement (e.g. ensuring the integrity of the register stack for IA64 code) but it allows minimal interferences with both the behavior and performance of instrumented applications. Instrumented generates an instrumented_assembly code.
- *Printer:* applies what the user wants to do. That can be done on batch mode or interactive mode. It is necessary to choose the level and the mode of instrumentation. Printer is designed to provide:
  - An instrumented_companion file, is a C code that allocate a structures and fill them with data that user fixed in input printer.
  - An instrumented_Makefile, loads flags compilation to compile instrumented_assembly code.
  - An instrumented_wrapper, add the new files generated by previous submodules to main program.
  - the only thing user has to do is unpack the tgz, to type make (or gmake) and to linked the wrapped main.
- *Execution:* an execution of makefile produce by printer, generates an instrumentation results. At this stage, the instrumenter results are not stored in the common database from further analysis, but are printed in mir files (mir: MAQAO Instrumentation Results).
- LUA script: parses the mir files and reproduce a structured representation saved in MAQAO database. In this level, MAQAO combine static and dynamic analysis, answers users quires, add user interface possibilities and reload results instrumentation.

The principal results are:

- Posted on CFG. The CFG printer, is used to generate hotpath computation. Hotpath results are reloaded from data bases when reloading an assembly for which .mir has already been parsed.
- Summarized in performance advisor.
- Summarized in new windows in MAQAO interface.
- To post the results of the instrumentation in the code assembler in an explicit way.

The three previous points are generated with an *internal interpreter script.*

5. MAQAOPROFILE Architecture

MAQAO proceeds to code instrumentation automatically. This is done by injecting a limited number of extra-bundles, named *assembly probes*, around the targeted code fragments to monitor. These bundles are in charge of storing some

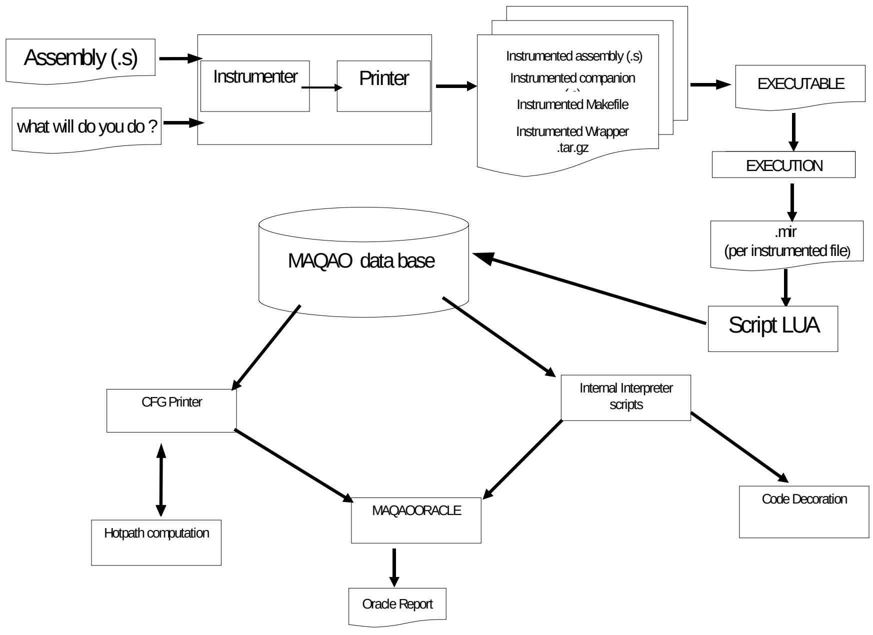


Figure 5. The MAQAOPROFILE process

specific registers in a dedicated memory zone. MAQAO ensures that the register stack remained unchanged by the instrumentation (by analyzing allocated registers or by adding spill/fill instructions). This low level technique allows to minimize the main instrumentation drawback: instrumentation code altering the compiler code generation and optimization chain behavior. Therefore MAQAO measures the real application behavior with minimal disturbance. An interesting side effect of low level instrumentation is its very low run-time overhead.

MAQAO instrumentation can be done at function level, loop level, basic block level or instruction level. For all these levels of instrumentation MAQAO monitors and stores the value of the clock register and builds execution time profile. Additionally MAQAO is able to store every register value manipulated by the code and dynamically process it. Monitoring value manipulated by a binary at run time is often reffered to as *value profiling* [9].

5.1 Time Profiling: Hot Path

For instrumentation done at basic block level, we have found only one valuable usage: Hot path computation. Therefore we specialize this level, and basic block instrumentation is dedicated to production of hot paths. While classic tools such as `gprof` can isolate the most important function, its scope is too narrow, limited to almost the notion of function. Identifying the path at run-time which crosses the whole program where the application spends the most of its time is a key element for understanding application behavior [10], [11].

Hotpath is based on the classic Bellman's algorithm (BA) [12]. Basically hot path computation is done in five steps:

- Basic-Block instrumentation
- Instrumented code execution

- Dynamic results injection in the CFG and set-up for BA.
- Force BA to respect evaluation of function head (call site independence)
- Execute again BA with path respecting function call/return (which means managing a call stack)

Step 1: Assembly probes are injected at the entry/exit of every basic blocks.

Step 2: Thus at run-time, a counter is incremented each time the corresponding edge of the CFG is taken.

Step 3: After the execution stage, once all counters are available. Control flow graphs for all code's functions are unified (without duplicating a function graph, is the function is called several times). Back edges for loops are deleted to avoid to trap the Bellman's algorithm and to save computation time as BA's complexity could approach $O(n^3)$ [2].

Step 4: Once the CFG for the whole code is produced with all edges annotated by their respective counter, return edges are weighed correspondingly to the edge between the calling block and the return block. These edges are then deleted to force Bellman's algorithm to go through the function. Then the Bellman's algorithm is used to update the evaluation of the vertex (and not edges) An extension will be to evaluate vertex, for instance according to the latency in order to compute the most expensive path, or to the number of instruction to compute the largest path), but currently they are set to 1 as we focus on frequently used pathes.

However the first basic block of each function as an evaluation set to 0, this prevents the algorithm to stumble on call site. Therefore an understated assumption is that a function could be considered as an edge with an evaluation equal to the sum of the evaluation of its hot path (like introducing a delay due to its hot path). This can be seen as a recursive hot path computation.

Step 5: Finally the critical path is parsed respecting function calls and returns. Taking advantage of this stage, multiple-exit loops are examined and graph exploration is pushed up to the point where the same edge is never taken twice. Result of this procedure is depicted in Figure 6.

Instrumentation Data storage

Once the technical challenge of gathering data dynamically has been fulfilled, an interesting question is how to extract meanings from this vast amount of information. MAQAO supports two forms for data storage:

- *dispersion sensitive*: any new data are dynamically compared to the previously stored data. From this comparison data are put in the corresponding histogram bucket. For implementation reason bucket sizes are powers of two. This allows to build an accurate picture of data dispersion which is crucial to apply specialization techniques.
- *order sensitive*: the previously described method implicitly discards part of the dynamic information. In fact, building histogram drops the notion of sequence and execution order. Therefore, the second storage method is a rotating buffer where data are stored in the order in which they appear. This is very important for instance to computed address variation and to detect stride memory accesses.

[2] Notice that currently our hot path algorithm does not support recursive function calls.

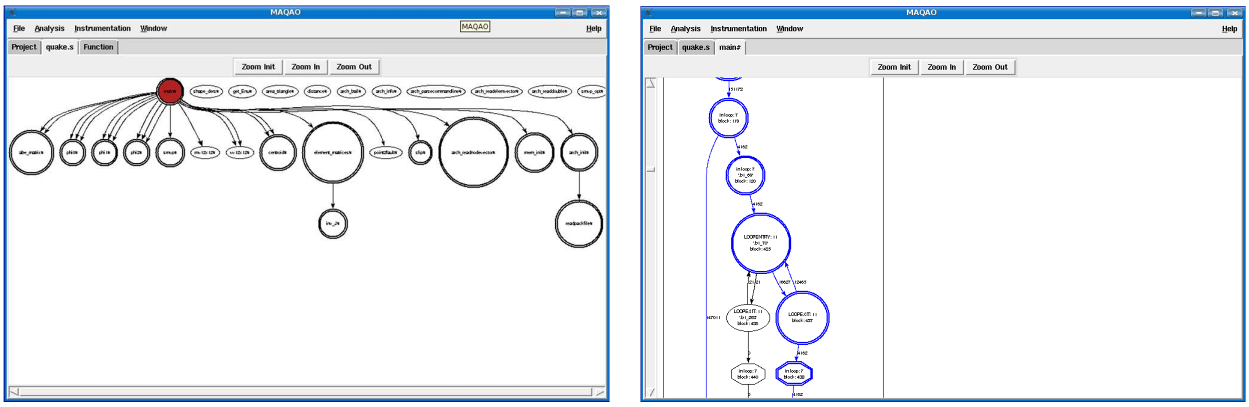

MAQAO
File Analysis Instrumentation Window Help
Project quake.s Function
Zoom Init Zoom In Zoom Out
MAQAO
File Analysis Instrumentation Window Help
Project quake.s main#
Zoom Init Zoom In Zoom Out

Figure 6. Displaying Hotpath information in MAQAO for 183.equake CG and a sample of its CFG.

5.2 Value Profiling

A key feature of MAQAO is its ability to value profile the code at various granularity. Value profiling is often the missing link between the observed behavior on the hardware and the nature of the application. This feature yields to numerous optimization opportunities.

5.3 Instrumentation at Function level

As detailed in section 1.1.2 detecting an always strictly positive parameter was the critical point to improve FFTW4 codelet by more than 40%. From MAQAO GUI it is easy to select a function, or to select all application functions and monitor their parameters (either in dispersion or order sensitive mode).

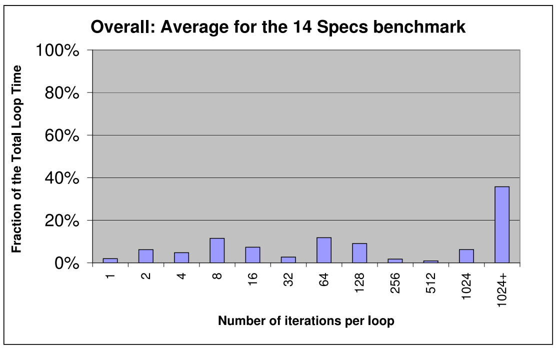
5.4 Instrumentation at Loop level

Figure 7 details results gathered by instrumentation at loop level of the SPEC FP 2000 benchmarks. We use the compilations flags (`-fast`, `-fno-alias` and so on) as reported by vendors on Spec website [13] except that inter-procedural optimization were disabled (`no-ipo`).

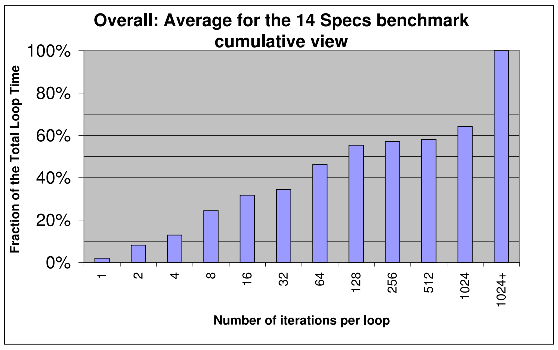
These numbers [3] provide insight about code specialization opportunities for short loops where software pipeline and other unrolling techniques are often low-performers. In fact, this is specially important since low-level optimization are always targeting the asymptotic performance, neglecting all start-up effect as cold start. For a short trip, start-up cost can be the dominant one.

From a performance analysis point of view, these numbers are taking advantage of both time and value profiling. Time profiling allows to give a precise weight to all executed loops, therefore underscoring hotspot. Value profiling monitors the iteration count. Correlating this information provides the relevant metric: i.e. which hot loops are short. This a clear illustration of the interest of centralized approach for performance analysis.

[3] During our experiments we have to mention that two codes run into problems within MAQAO instrumentation: 178.galgel die during execution (after roughly a quarter of its supposed execution time). And 188.ammp is re-entrant and our instrumentation is not very robust confronted to recursivity. Nevertheless, we chose to take their results in consideration since they were consistent with experiments conducted by other means.



(a) SPEC FP overall loop distribution



(b) SPEC FP overall cumulative loop distribution

Figure 7. Loop Execution Time depending on loops number of iterations. These graphs depicts the execution time distribution per loop depending on the total number of iterations. Histograms summarize iterations weight for an interval in power of two:]0,1],]1,2],]2,4],]4,8] and so on. For instance the bar labeled 64 coalesces all loops with a total number of iterations between]33,64]. These two figures summarize results obtained on the whole SPEC FP benchmarks suite. 7(a) depicts average on a per benchmark basis, i.e. each benchmark is considered individually without weight related to its execution time. 7(b) presents the same data but in a cumulative histogram which is convenient to grab the slope of loop distribution. From these two graphs it can be said than Spec benchmarks spend 25% of their loop time within loops of less than 8 iterations.

5.5 Instrumentation at Instruction level

Knowing dispersion of value manipulated by each instruction can be valuable to take optimization related decision. For instance, characterizing address streams allows to detect bank conflict or aliasing problem. Observing that a given value is almost always constant can justify the cost of high level code specialization.

MAQAO currently supports the following rules: automatic detection of prefetch distance.

For prefetch distance estimation, the algorithm is straightforward: all the instructions within the loop are instrumented. Data are stored in order sensitive way (rotating buffer).

1. Considering the address flow from the prefetch instruction: `0xf000 0xe000 0xf010 0xe010`
2. Considering the address flow from a load instruction: `0xef00 0xef10 0xef20 0xef30 0xef40 0xef50 0xef60 0xef70 0xef80 0xef90 0xefa0 0xefb0 0xefc0 0xefd0 0xefe0 0xeff00xf000`
3. Proceed to pattern matching for the first prefetch address. Since both instructions are within the same loop they are executed jointly (we do not handle case with if conversion). The first matching address in the load address flow (if any) returns the prefech distance. For instance, here data are prefetched 16 iterations ahead.

Evaluation of the prefetch distance can also be done statically based on DDG. However, at the opposite of the static module, instruction instrumentation handles data stream interleaving (an optimization used Itanium by *icc* [14] to fetch alternatively two data streams with a single prefetch instruction)

6. MAQAORACLE: Performance Advisor

ICC outputs an optimization report, providing information on the success/failure of each optimization phase. This report can be considered as an execution trace of the optimization process. The report neither gives any clue on how to cope with some optimization issues, nor points out what are these issues. It just plainly states which optimization took place, where, and with which value of parameter. The goal is therefore different from MAQAO. However, it is interesting to make a comparison between information collected by both tools. MAQAO provides an analysis of this data while the optimization report only reports raw data.

6.1 Performance Oracle

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. It is possible to interpret performance data in numerous ways... and to be lost. The *Performance Oracle* is built over a set of rules and metrics and act 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 [15]. However ATExpert was focused on parallelization issue and was neither as extensible nor as sophisticated as MAQAO's performance module.

Oracles rules are either purely based on static analysis or use hardware counter and instrumentation information. All rules are written in LUA [16] with the support of a MAQAO API. This C/LUA API allows to manipulate MAQAO internal program representation and to write quickly compact rules. Thus, the Oracle is a library of high level rules which can be extended according to user needs.

6.2 Hierarchical Reporting Approach

A classic pitfall for reporting tools is to overload end-user under a mountain of data. Therefore, this trend leads to miss the initial goal which is to help in the decision process. The needs of 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, MAQAO 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 analysis or whole code analysis. Additionnaly a filter can be set depending on the degree of confidence of Oracle answer or the potential performance gain involved.

- The first and most exhaustive level is the instructions level view. For each loop, selected instructions counts and built-in metrics are displayed. These counts 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. Instruction are coalesced per family (such as integer arithmetic, loads instructions and so on) and counted on a per basic block basis. However the goal is not to catch dispersion rules (hence the taxonomy) of the architecture, but to detail instructions that have been determined as being of special interest. This instruction count is enriched by built-in metrics : *Cycle cost per iteration*, *issue cost* *per iteration*, *Theoretical cycle bounds per iteration*. Together counts and metrics are exploited by Performance Oracle rules. Rules also process results gathered during application execution (instrumentation, hardware counter, cycle count).

- The second level, which is already an abstraction layer, only reports loop where some important performance features are detected, thus is filtering out a large amount of non-essential data. Additionally results are reported in a user-friendly way (not just a bunch of number but clear sentences!)
- The third and fourth levels are summary tables. Whereas 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 ease comparison.

6.3 Example of Oracle Output

Here is an example of Performance Oracle output for a loop: Corresponding source code is:

$$\left.\begin{array}{ll} DO & I=1,NPSTACK \\ & ZZ(I,K)=0 \\ END & DO \end{array}\right\}\text{should be replaced by a call to }\texttt{memset}$$

Results Sample [code M - function Chociso]:

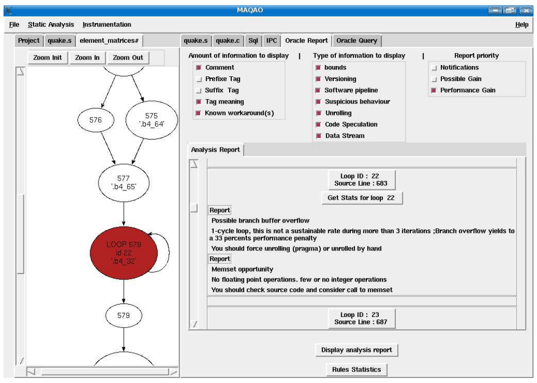
```
=> LOOP SOURCE LINE  426  ASSEMBLY VERSION 2 [MAQAOracle internal loop. id  265 ]
     MAQAO report: No floating point operations. few or no integer operation.
     check source code and consider call to memset.
     MAQAO report: Scheduling is matching optimal bound.
     This is a clear sign of code quality but not a proof (beware of code bloating).
     MAQAO report: Analysis hints unroll factor of 8
```

Performance Oracle results are displayed in the MAQAO interface. In front of each loop of the source code, ⓘ gives access to the information computed by the Oracle concerning this loop. Note that several ⓘ in front of the same loop means that the function of the loop has been inlined several times. In interactive mode MAQAO can give different advices according to the inlined version, since the compiler may have changed the optimizations according to the inlining site.

Note on Loop fusion: this high level loop transformation is difficult to detect at the assembly level. The list of fused loops is given by the source line number of the fused loops. This information is not found by MAQAO since it would need a detailed analysis of the source code. Therefore loop fusion is not caught by the Performance Oracle.

7. Related Work

Two main classes of low level instrumentation tools can be related to MAQAO. One class is composed of performance analysis tools aiming at understanding application behavior based on hardware counters. Fall in this category tools such as VTune (self-contained program), or PAPI (user-independent). The other family of tools is more focused on code



Amount of information to display: ■ Comment, ☐ Prefixe Tag, ☐ Suffix Tag, ■ Tag meaning, ■ Known workaround(s)

Type of information to display: ■ bounds, ■ Versioning, ■ Software pipeline, ■ Suspicious behaviour, ■ Unrolling, ■ Code Speculation, ■ Data Stream

Report priority: ☐ Notifications, ☐ Possible Gain, ■ Performance Gain

Analysis Report

Loop ID : 22
Source Line : 683

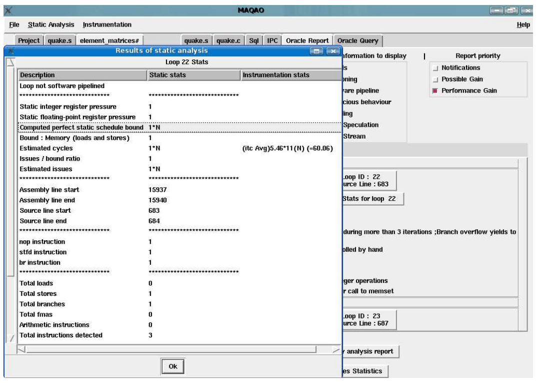
Get Stats for loop 22

Report
Possible branch buffer overflow
1-cycle loop, this is not a sustainable rate during more than 3 iterations ;Branch overflow yields to a 33 percents performance penalty
You should force unrolling (pragma) or unrolled by hand

Report
Memset opportunity
No floating point operations. few or no integer operations
You should check source code and consider call to memset

Loop ID : 23
Source Line : 687

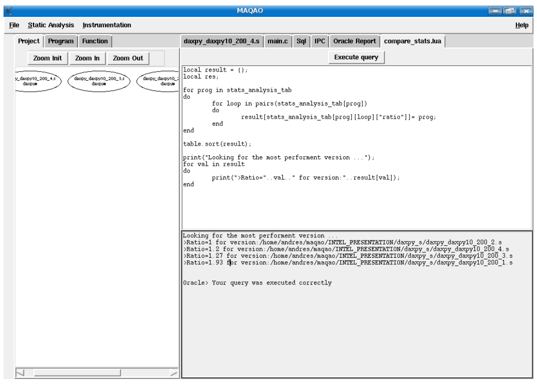
(a) Loop selected in the CFG: Performance Oracle displays analysis. The loop is an one-cycle loop and is similar to a memset routine. Thus it should be replaced by a call to the hopefully optimized memset routine



Loop 22 Stats

Description	Static stats	Instrumentation stats
Loop not software pipelined		
******************************	******************************	
Static integer register pressure	1	
Static floating-point register pressure	1	
Computed perfect static schedule bound	1*N	
Bound : Memory (loads and stores)	1	
Estimated cycles	1*N	(itc Avg)5.46*11(N) (=60.06)
Issues / bound ratio	1	
Estimated issues	1*N	
******************************	******************************	
Assembly line start	15937	
Assembly line end	15940	
Source line start	683	
Source line end	684	
******************************	******************************	
nop instruction	1	
stfd instruction	1	
br instruction	1	
******************************	******************************	
Total loads	0	
Total stores	1	
Total branches	1	
Total fmas	0	
Arithmetic instructions	0	
Total instructions detected	3	

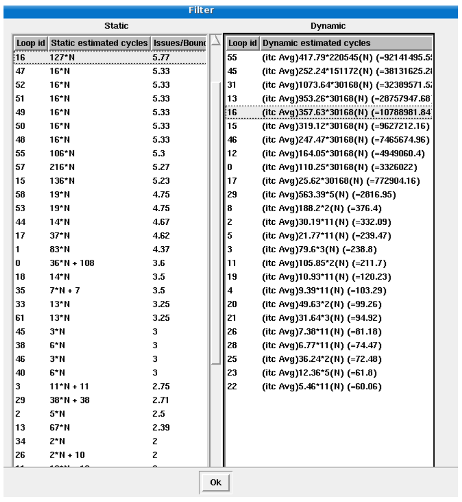
(b) Analysis of the loop with inclusion of dynamic information. Performance is bounded by Memory. While the perfect static schedule is estimated to 1 cycle per Iteration instrumentation results show that the average loop trip count is 11 and the cost 5.5 cycle per iteration. Static estimation is exceeded by a factor of 5.



```
local result = {};
local res;

for prog in stats_analysis_tab
do
        for loop in pairs(stats_analysis_tab[prog])
        do
                result[stats_analysis_tab[prog][loop]["ratio"]]= prog;
        end
end

table.sort(result);

print("Looking for the most performent version ...");
for val in result
do
        print(">Ratio="..val.." for version:"..result[val]);
end
```

```
Looking for the most performent version ...
>Ratio=1 for version:/home/andres/maqao/INTEL_PRESENTATION/daxpy_s/daxpy_daxpy10_200_2.s
>Ratio=1.2 for version:/home/andres/maqao/INTEL_PRESENTATION/daxpy_s/daxpy_daxpy10_200_4.s
>Ratio=1.27 for version:/home/andres/maqao/INTEL_PRESENTATION/daxpy_s/daxpy_daxpy10_200_3.s
>Ratio=1.93 for version:/home/andres/maqao/INTEL_PRESENTATION/daxpy_s/daxpy_daxpy10_200_1.s


Oracle> Your query was executed correctly
```

(c) Writing a rule in LUA to analyze the whole code behavior. One may be interested in knowing 1/ the hot loops and 2/ if hot loops are far from their static schedule. Such request can be easily expressed in LUA. This language embedded in MAQAO allows to access all the internal data structure which are a repository of code performance information.



Static

Loop id	Static estimated cycles	Issues/Bounc
16	127*N	5.77
47	16*N	5.33
52	16*N	5.33
51	16*N	5.33
49	16*N	5.33
50	16*N	5.33
48	16*N	5.33
55	106*N	5.3
57	216*N	5.27
15	136*N	5.23
58	19*N	4.75
53	19*N	4.75
44	14*N	4.67
17	37*N	4.62
1	83*N	4.37
0	36*N + 108	3.6
18	14*N	3.5
35	7*N + 7	3.5
33	13*N	3.25
61	13*N	3.25
45	3*N	3
38	6*N	3
46	3*N	3
40	6*N	3
3	11*N + 11	2.75
29	38*N + 38	2.71
2	5*N	2.5
13	67*N	2.39
34	2*N	2
26	2*N + 10	2

Dynamic

Loop id	Dynamic estimated cycles
55	(itc Avg)417.79*220545(N) (=92141495.5
45	(itc Avg)252.24*151172(N) (=38131625.2
31	(itc Avg)1073.64*30168(N) (=32389571.5
13	(itc Avg)953.26*30168(N) (=28757947.68
16	(itc Avg)357.63*30168(N) (=10788981.84
15	(itc Avg)319.12*30168(N) (=9627212.16)
46	(itc Avg)247.47*30168(N) (=7465674.96)
12	(itc Avg)164.05*30168(N) (=4949060.4)
0	(itc Avg)110.25*30168(N) (=3326022)
17	(itc Avg)25.62*30168(N) (=772904.16)
29	(itc Avg)563.39*5(N) (=2816.95)
8	(itc Avg)188.2*2(N) (=376.4)
2	(itc Avg)30.19*11(N) (=332.09)
5	(itc Avg)21.77*11(N) (=239.47)
3	(itc Avg)79.6*3(N) (=238.8)
11	(itc Avg)105.85*2(N) (=211.7)
19	(itc Avg)10.93*11(N) (=120.23)
4	(itc Avg)9.39*11(N) (=103.29)
20	(itc Avg)49.63*2(N) (=99.26)
21	(itc Avg)31.64*3(N) (=94.92)
26	(itc Avg)7.38*11(N) (=81.18)
28	(itc Avg)6.77*11(N) (=74.47)
25	(itc Avg)36.24*2(N) (=72.48)
23	(itc Avg)12.36*5(N) (=61.8)
22	(itc Avg)5.46*11(N) (=60.06)

(d) Rule result: full comparison with static and dynamic performance including hot loops profiling information. Loops are sorted by decreasing weight and dynamic and static performance are compared. From a pure static evaluation loop 16 as a static schedule of 127 cycles per iteration which is 5 times the value of its optimal bound. Additionally it appears that dynamic cost per iteration is 357 cycles!. Clearly this loop worth investigation.

Figure 8. Example of Oracle analysis for a selected loop from the 183.equake CFG.

manipulation like Salto, or code instrumentation such as ATOM or PIN. However MAQAO broad approach is more related to the path chosen by HPCview or to a lesser extend by Fineesse, this later one being mor focused on parallelization than code optimization.

Hardware monitors are extremely helpful for performance tuning, they are the backbone of analysis tools like `VTune` [17], `Caliper` [18], `Cprof` [19]. Their usage is so widespread that an API gets standardized to describe their access [3]. Nevertheless, hardware counters are limited to the dynamic description of an application and this picture needs to be correlated with other metrics. For instance, from the hardware counter point of view, code bloating (or even dead code) filling up functional units and leading to high IPC is seen as a desirable behavior.

On the static side, `Salto` [20] is a framework dedicated to the implementation of complex assembly code transformations. After being parsed, assembly code is seen as a collection of C++ objects plugged in a user-developed application. `Salto` is more a toolkit than a tool and could appear as a back-end of MAQAO diagnosis chain: once a problem is identified by MAQAO, some transformations have to be applied by SALTO to solve this problem. DPCL [21] (Dynamic Probe Class Library) is a set of C++ classes from IBM originally based on Dyninst [22]. The purpose is to help developers to support dynamic instrumentation of parallel jobs. Probes can be inserted in a running binary to check the hardware counters or cycles for any function of the monitored code. Even if dynamic instrumentation is very appealing, DPCL does not include any notion of code inspection.

`ATOM` [23] (for Alpha assembly) and `Pin` [24] (for Intel architectures assembly) 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 (Executable Editing Library) [25] belongs to the same categories of tools. This C++ library allows to edit a binary and add code fragment on edges of dissambled 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.

`HPCview` [26] and `Finesse` [27] (this one being more oriented toward parallelization) address the analysis problem from static and dynamic sides. `HPCview` tackles the same problem as MAQAO: the complex interaction between source code, assembly, performance and hardware monitors. `HPCview` presents a well designed GUI based on web browser, displaying simultaneous views of source, assembly code and dynamic information. This interface is connected to a database storing for each statement of the assembly code a summary of its dynamic information. Based on control flow graph and a tool named `bloop`, `HPCview` builds abstracted representation of code loop structures (using an XML interface). Some important differences should be underscored: while a database is embedded in the application, end-user has only limited opportunity to explore the code and define new queries. `HPCview` also lacks value profiling which can lead to powerful, yet simple to implement optimizations such as code versioning.

Shark [28], [29], developed by Apple offers a comprehensive interface for performance problem. As 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 (including call stack, garbage collection, binary analysis), and it underlines the need to think performance software beyond gprof.

8. Conclusion

With MAQAOPROFILE, our MAQAO instrumenter, we have proven when the instrumenter is merged as MAQAO module can providing more powerful analytic capabilities than existing tools. MAQAO is able to drive the instrumenter and to take advantage of dynamic behavior results. By combining static and dynamic analysis allows to refine quickly code performance analysis and offers capabilities complementary to the standard performance counter based tools.

MAQAOPROFILE's ability to produce different result level enables user to choose the granularity and to define the lag he wants to introduce in his code. Project context is able to perform analysis on multi sources and to instrument those project. Based on MAQAOPROFILE and MAQAORACLE, the performance analysis methodology proposed allows to perform efficient code optimization. Currently MAQAO parses codes generated by different compilers (IA32, IA64, x86), but MAQAOPROFILE is limited to itanium architecture. We are working to port MAQAOPROFILE to x86. We hope MAQAO will continue to be an effective platform for embedded architecture. Another architectures, but more prospective, is the ST200 and TriMedia.

In short terms the GUI will be re-designed as a Web interface. The idea is to go toward a client/server architecture, where MAQAO would be executed remotely and driven through a web browser. More interface work is also planned for a smarter support of hardware counters.

References

[1] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J-T. Acquaviva, W. Jalby MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2 In *Workshop on EPIC architectures and compiler technology*, San Jose, 2005.

[2] Stéphane Eranian, Perfmon project home page: www.hpl.hp.com/research/linux/perfmon HP Labs

[3] Jack Dongarra, Kevin S. London, Shirley Moore, Philip Mucci, Daniel Terpstra, Haihang You, Min Zhou. Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters. IPDPS 2003: 289

[4] Montecito Processor wikipedia http://en.wikipedia.org/wiki/Montecito_(processor)

[5] Matteo Frigo and Steven G. Johnson, The Design and Implementation of FFTW3, Proceedings of the IEEE 93 (2), 216-231 (2005), Special Issue on Program Generation, Optimization, and Platform Adaptation, www.fftw.org

[6] *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*, revision 2.1 edition. http://developer.intel.com/design/itanium/family.

[7] Intel Itanium2 Processor Reference Manual for Software Development and Optimization, *http://download.intel.com/design/Itanium2/*

[8] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J-T. Acquaviva, W. Jalby Exploring Application Performance: a New Tool for a Static/Dynamic Approach In *Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, 2005.

[9] B. Calder, P. Feller and A. Eustace Value Profiling, Proceedings of Micro-30, December 1-3, 1997, Research Triangle, North Carolina

[10] James R. Larus, Whole program paths, PLDI ’99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, 1999, pages259–269,

[11] Gilles Pokam and François Bodin, An Offline Approach for Whole-Program Paths Analysis Using Suffix Arrays, LCPC, 2004, pages 363–378

[12] Richard Bellman, On a Routing Problem Quarterly of Applied Mathematics 16(1), pp.87-90, 1958

[13] Standard Performance Evaluation Corporation, *http://www.spec.org/cpu2000/results/res2006q2/*

[14] G. Doshi and R. Krishnaiyer and K. Muthukumar Optimizing software data prefetches with rotating registers *2001 International Conference on Parallel Architectures and Compilation Techniques* 2001, Barcelona, Catalunya, Spain

[15] John Kohn and Winifred Williams, ATExpert, Journal of Parallel and Distributed Computing 1993 vol. 18 Issue: 2, p. 205–222

[16] Roberto Ierusalimschy and Luiz Henrique de Figueiredo and Waldemar Celes Filho, *Lua — an Extensible Extension Language*, ”Software Practice and Experience”, 26(6):635–652, june 1996. http://www.lua.org

[17] Intel Corporation. VTune Performance Analyzer http://www.intel.com/software/products/vtune

[18] Robert Hundt, HP Caliper: An Architecture for Performance Analysis Tools, Proceedings of the First Workshop on Industrial Experiences with Systems Software, WIESS 2000, October, 2000, San Diego, CA, USA. USENIX 2000 http://www.hp.com/go/caliper

[19] http://sourceforge.net/projects/cprof

[20] Erven Rohou, François Bodin, Andre Seznec, Gwendal Le Fol, Francois Charot and Frederic Raimbault. SALTO : System for Assembly-Language Transformation and Optimization. RR-2980, 27 p., citeseer.ist.psu.edu/rohou96salto.html

[21] Luiz De Rose, Ted Hoover Jr. and Jeffrey K. Hollingsworth, The Dynamic Probe Class Library: An Infrastructure for Developing Instrumentation for Performance Tools, www.ptools.org/projects/dpcl IPDPS 2001: 66

[22] B. R. Buck and J. K. Hollingsworth, An API for runtime code patching Journal of High Performance Computing Application, 14(4):317-329, 1994.

[23] Amitabh Srivastava and Alan Eustace. ATOM - A System for Building Customized Program Analysis Tools. PLDI 1994: 196-205

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation Micro 37, Portland, OR., 2004

[25] James R. Larus and Eric Schnaar. EEL: Machine-Independent Executable Editing, *appeared in the ACM SIGPLAN PLDI Conference*, June 1995.

[26] J. Mellor-Crummey, R. Fowler and G. Marin. HPCView: A tool for top-down analysis of node performance. Computer Science Institute Second Annual Symposium, Santa Fe, NM, October 2001. 2001, citeseer.ist.psu.edu/mellor-crummey01hpcview.html http://hipersoft.cs.rice.edu/hpctoolkit/papers.html

[27] N. Mukherjee, G.D. Riley and J.R. Gurd. FINESSE: A Prototype Feedback-guided Performance Enhancement System. Parallel and Distributed Processing (PDP) 2000, Rhodes, Greece, January 2000

[28] Optimizing Your Application with Shark 4 http://developer.apple.com/tools/shark_optimize.html

[29] Optimize with Shark: Big Payoff, Small Effort http://developer.apple.com/tools/sharkoptimize.html