Exploring Application Performance: a New Tool For a Static/Dynamic Approach

Lamia Djoudi[1] Denis Barthou[2] Patrick Carribault[3,1] Christophe Lemuet[1]
Jean-Thomas Acquaviva[1] William Jalby [1]

[1] LRC ITACA, CEA/DAM and Université de Versailles Saint-Quentin, France
[2] PR*i*SM, Université de Versailles Saint-Quentin, France
[3] Bull SA, Les Clayes sous Bois, France

Abstract

Application performance is highly dependent upon the quality of assembly code produced by the compiler, this trend is further reinforced on EPIC architecture. Therefore assessing precisely the quality of the generated code is essential to deliver high performance. Nowadays performance troubleshooting is mostly tackled by using hardware counters and dynamic profiling. In this paper we propose a tool for performing analyses of assembly code (including some runtime capabilities). We advocate that such a tool can be combined with classical hardware counter analysis to achieve similar results at much lower cost and better accuracy than an approach relying only on hardware performance counters.

Among the distinctive key advantages that our tool MAQAO offers: versatility (user can specify his own analyses) and precise and fast diagnosis capabilities. These capabilities enable a better control of an optimization process and enhance the productivity of programmers in the process of code tuning.

Performance tuning driven by MAQAO on two real codes allows substantial gains in execution time (up to a factor of 2). Optimization strategies and potential hotspots are highlighted by a joint usage of static and dynamic analyses. Relying on MAQAO allows to leverage the somehow complex task of understanding performance bottlenecks. Clear details of this process are provided for both codes.

1 Introduction

Quality of the code produced by the compiler is essential to get high performance. In the old CISC days, quality could be simply assessed by counting the number of instructions. Nowadays, with the recent generation of microprocessors, such simple metrics are no longer valid. First of all, caches have introduced data locality as a key performance metric. Code quality also results from the appropriate use of instructions such as branches, fused multiply add `-fma-`, predicated instructions or prefetches. Finally, some low level architecture mechanisms such as bank conflicts or load/store queue aliasing have an impact on execution time and have to be taken into account by code generators.

In fact, on modern microprocessors, taking into account all the architectural aspects is critical to get high performance. For example on Itanium systems, *gcc* is very often outperformed by Intel C Compiler (*icc*) because *icc* considers all of the features offered by the Itanium 2 [19, 20]. However this performance search comes at the expense of code complexity and stability: the code generated is difficult to analyze and optimizations are often unstable, meaning that a slight modification on the source code or on some input values, may change drastically the overall performance. In particular, most of the optimization process relies on heuristics whose exact behaviors are hard to predict and sometimes, turning on all of the optimization flags results in a code which is slower than just turning a limited set of optimizations. Therefore, it is critical to be able to analyze the quality of the code produced.

Performance analysis has made tremendous progress with the appearance of low level hardware counters capable of tracking various events. Such counters are extremely helpful to locate performance bottlenecks: for example poor data locality automatically generates high cache miss ratio which can be easily captured. However dynamic behavior analysis does not allow to discover all of the potential performance problems. For example, missed *standard* optimizations such as constant propagation or common subexpression elimination are much more easily detected on the assembly code than by hardware performance counters.

Consequently we propose a new tool, named MAQAO (*Modular Assembly Quality Analyzer and Optimizer*), able to tackle what we saw as the real problem: *inspection/analysis* of assembly code. Our goals can be summarized by the following:

- Intelligent navigation and flexible automated analysis (either predefined or user defined) of assembly code;
- Quality assessment of the code generated and detec-

tion of potential inefficiencies of the assembly code either statically or dynamically (value profiling);

- Providing hints and guidelines to drive optimization process;
- Build an evolving database of known performance issues on the target architecture (currently Itanium 2).

The major contributions of MAQAO are:

- To provide the user with a tool for performing flexible quality analysis and a methodology on code performance. Quality assessment relies on advanced pattern matching where patterns have been identified either by micro-benchmarking techniques [3] or by previous expertise. This is addressing a problem where currently most of the optimization knowledge is informally stored (if not only in the programmer's brain). This productivity issue is not handled by other tools.
- To store both high level code structures (loops, CFG) and low level profiling information (hardware counters, value profiling) in a database to perform requests on it. This is an important and novel aspect of code analysis. MAQAO therefore proposes the first building blocks of an assembly optimizer/analyzer.
- To provide value profiling for addressing the link (missing in other tools) between observed behavior on the processor and software behavior.

EPIC architecture is the first target for MAQAO due to its in-order execution and explicit parallelism: there is no hardware mechanism to smooth scheduling problems, instruction selection and so on. Still MAQAO could be ported to another architecture. MAQAO parses assembly files and generates a structured representation. The structures built are the call graph, the control flow graph, the loop structures and the dependence graph (for registers). They are inserted into an SQL database that is later used for static analysis. Queries to the database can detect `fma` or compute pipelined loop latencies. Structuring assembly code is not only useful for queries but also enables to modify it. A first application of this ability is insertion of performance probes (instrumentation) directly at the assembly level for dynamic analysis.

Section 2 provides a motivating example advocating the use of static analysis compared to dynamic hardware counters. Section 3 gives an overview of various related performance analysis tools. Section 4 describes the MAQAO static analysis phase, how the assembly code is structured and how pattern detection can be performed. Section 5 details how to compare performance bounds to the code produced by the compiler. Section 6 presents how code instrumentation is handled by MAQAO. Section 7 provides an overview of the methodology used with MAQAO to optimize codes. Section 8 illustrates how the diagnostics produced by MAQAO helped us to optimize two real life codes: a scientific code called TERA, and SHA-0 attack, a cryptanalysis application. Finally, conclusion and perspectives are provided in Section 9.

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.

fmas	28
fa	16
fm	6
loads	14
stores	8
branch	1
instructions/iteration	120
cycles/iteration	28
swp	no swp
stalls	5

Table 2. Static analysis of instructions in the FFTW 4 loop. Numbers are for one iteration. `fma` corresponds to the total number of floating point multiply, add and multiply add while `fa` (resp. `fm`) corresponds to the real number of simple floating point add (resp. multiply). Stalls are predicted by compiler and are given through comment lines.

2 Motivating example

FFTW (Fastest Fourier Transform of the West) [23] is a highly tuned version of the FFT. Most of the computation occurs within small code pieces named *codelet* (currently fftw_twiddle). 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.

Results from Table 1 depict a code with a very good IPC and a rather limited fraction of stall cycles. However, this does not expose the code bloating problem, i.e. a question remains unanswered: among all the issued instructions, how many are *essential/useful* instructions and how far is this code from an optimal one? A glance at some static metric collected via MAQAO on the assembly code depicts the following picture (see Table 2).

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).

Notice that Itanium hardware counters are able to detect if an `fma` instruction corresponds to a real fused multiply add or to a simple multiply or simple add. Therefore the 6 real fused operations are reported by the hardware as being 12 FP_OPS, while the 22 remaining operations are just counted once. Hence for every loop trip counters report 12+22 = 34 floating point operations which corresponds exactly to the value found by MAQAO.

Stalls are given 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 $\ldots$). The compiler predicts 5 stall cycles per iteration while the hardware counter measured 9 stall cycles per iteration. If we add this 4 extra cycles to the number of cycles estimated by the compiler, this 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: inspection of assembly code shows that the loop is neither pipelined nor unrolled (this is checked by comparing the number of real multiplications in the assembly and in the source). Furthermore, MAQAO reveals that the 28 fmas could be executed in 14 cycles and that the loads/store instructions could be executed in 6 cycles, therefore the lower bound (due to resource constraints) is 14 cycles per iteration which is far away from the 28 cycles per iteration (a potential factor of 2 is achievable). Therefore, the instruction schedule is fairly poor.

A 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 than exposed in Table 1 (backed by the static analysis), results are detailed in Table 3. This time, the compiler generates a far better schedule and pipelines the loop. Note that the speed-up of two is nearly reached for large enough loop trip counts.

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. A stage of static analysis delivers interesting results, and is a shortcut to optimizations w.r.t. the clear but long way paved by hardware counters analysis.

loop trip	2	4	8	16	32	64
CPU Cycle	65	109	229	339	620	1174
Speed-up vs Table 1	1.33	1.37	1.21	1.61	1.71	1.78
Inst. issued	258	492	939	1448	2776	5432
F. OPs issued	68	136	272	544	1088	2176
Stall Cycle	10	8	44	52	119	215
Stall %	15%	7%	19%	15%	19%	18%

Table 3. Hardware counter measurements for FFTW 4 codelet. Improvement is greater than the number of stall cycle from the generic version.

3 Related Works

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. Our approach is a cross-over: assembly code inspection is done statically, data profiling is done using code instrumentation and hardware counters fit their traditional role of hotspot detection.

Hardware monitors are extremely helpful for performance tuning, they are the backbone of analysis tools like `VTune` [5], `Caliper` [18], `Cprof` [17]. Their usage is so widespread that an API gets standardized to describe their access [11]. 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` [7] 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 [24] (Dynamic Probe Class Library) is a set of C++ classes from IBM originally based on Dyninst [25]. 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 [4] (for Alpha assembly) and Pin [10] (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.

HPCview [8] and Finesse [9] (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.

4 Static Analysis

MAQAO parses any assembly code produced by *icc* and performs several static analyses on this code. As an important feature of the tool, the user has the possibility to widen the range of analyses and to quickly prototype new ones with a scripting module. From code parsing to loop detection, all analysis results are stored into a database associated with the analyzed program. This process is split in different phases detailed in this section.

4.1 Parsing Assembly

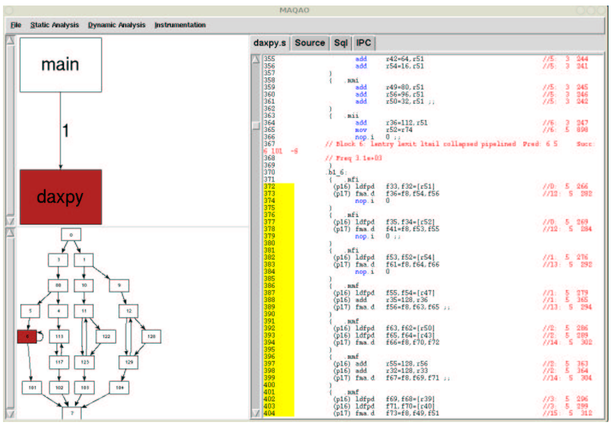
The instruction set reference of the Itanium Software Developer's Manual [21] describes IA64 instructions in more than 230 pages. We designed a script parsing automatically the manual in *pdf* form. This script generates the assembly code parser used by MAQAO, and with some light modifications, provided MAQAO front-end. Notice this process could be easily adapted to other architectures.

Instructions and assembly directives are stored in the database. Note that the generation of the original code from the data stored in the database is still possible. MAQAO has successfully parsed/analyzed codes of more than 250,000 lines in a few seconds on a desktop machine, a Pentium4 1.8GHz with 256MB.

The compiler provides some information concerning execution latencies (for each instruction issue), execution statistics for basic blocks (obtained after profiling) and gives the link between assembly and source lines. Although the parser does not complain if they are missing, all these values are also stored in the database.

4.2 Extracting the Structure from the Assembly

This section details the different analyses building the foundations for more complex ones. These analyses hierarchically restructure the assembly code, from functions down to basic blocks and bundles. An interactive interface provides access to these structures for the user (See Figure 1 for a snapshot), as well as a display of source and assembly codes and a scripting window for the tuning of new analysis. A batch mode also enables automatic analyses.



```
            add     r42=64,r51                         //5:   3  244
            add     r54=16,r51                         //5:   3  241
        }
        {   .mmi
            add     r49=80,r51                         //5:   3  245
            add     r56=96,r51                         //5:   3  246
            add     r50=32,r51 ;;                      //5:   3  242
        }
        {   .mii
            add     r36=112,r51                        //6:   3  247
            mov     r52=r74                            //6:   5  898
            nop.i   0 ;;
       // Block 6: lentry lexit ltail collapsed pipelined  Pred: 6 5     Succ:
6 101  -6
       // Freq 3.1e+03
        }
       .b1_6:
        {   .mfi
         (p16) ldfpd   f33,f32=[r51]                       //0:   5  266
         (p17) fma.d   f36=f8,f54,f56                      //12:  5  282
            nop.i   0
        }
        {   .mfi
         (p16) ldfpd   f35,f34=[r52]                       //0:   5  269
         (p17) fma.d   f41=f8,f53,f55                      //12:  5  284
            nop.i   0 ;;
        }
        {   .mfi
         (p16) ldfpd   f53,f52=[r54]                       //1:   5  276
         (p17) fma.d   f61=f8,f64,f66                      //13:  5  292
            nop.i   0
        }
        {   .mmf
         (p16) ldfpd   f55,f54=[r47]                       //1:   5  279
         (p16) add     r35=128,r36                         //1:   5  365
         (p17) fma.d   f56=f8,f63,f65 ;;                   //13:  5  294
        }
        {   .mmf
         (p16) ldfpd   f63,f62=[r50]                       //2:   5  286
         (p16) ldfpd   f65,f64=[r43]                       //2:   5  289
         (p17) fma.d   f66=f8,f70,f72                      //14:  5  302
        }
        {   .mmf
         (p16) add     r55=128,r56                         //2:   5  363
         (p16) add     r32=128,r33                         //2:   5  364
         (p17) fma.d   f67=f8,f69,f71 ;;                   //14:  5  304
        }
        {   .mmf
         (p16) ldfpd   f69,f68=[r39]                       //3:   5  296
         (p16) ldfpd   f71,f70=[r40]                       //3:   5  299
         (p17) fma.d   f73=f8,f49,f51                      //15:  5  312
```

Figure 1. MAQAO used on a daxpy routine. CFG is displayed, the selected basic block corresponds to the highlighted assembly code on the main window. Analyses can be selected from the above menus. Notice that for a simple daxpy function, the compiler has generated a complex control flow graph with several versions depending on the alignment of data on 16B boundaries.

The structure hierarchy of the assembly code is the following:

- *Call Graph*: this structure slices the code into different functions and shows how they interact each other. For a given function, the number of calls to another function is annotated on the edge going from the caller to the callee. The graph is represented with GraphViz [6], a toolkit ensuring a layout of the graph minimizing the intersection of edges, making it more manageable. Each node selection triggers the display of the selected function control flow graph.
- *Control Flow Graph*: the control flow graph structures basic blocks within a function, showing possible control paths. This information is important for loop detection and provides a quick assessment of

the function complexity. Selecting a node highlights the corresponding assembly code and positions the source code to matching lines. Likewise, selecting assembly or source code highlights the corresponding nodes. Notice that temporal pattern can be expressed in MAQAO, since the CFG and the sequential order inside basic blocks are kept in the database. This allow to express a relation of precedence between two instructions.

- *Dependence graph*: dependences are computed between statements w.r.t. registers (no memory location dependences) with a reaching definition analysis. This analysis is useful to validate code transformation and optimizations (still on-going work).
- *Loops*: the loops determine basic blocks that are likely to be the most executed. Hence, loops are a legitimate focus for analysis and optimization.
- *Bundles and Basic Blocks*: they are given by the compiler and are expected to be well-formed. Basic blocks are provided through comment lines but they could be computed if necessary.

Moreover, from the interface of MAQAO and thanks to the debugging information provided by the compiler, the user can navigate directly to and from assembly and source codes.

4.3 Implemented Analysis and Extensions

One of the key features of MAQAO is the possibility to express new analyses using scripts. This offers a wider range of analyses when standard statistics are not enough. Therefore, a knowledge base of interesting analyses can be built up from the experience of multiple users or from micro-benchmarking techniques[3]. The results of micro-benchmarks are patterns of code that do not perform well due to possible data alignment issues, conflicting memory banks or other memory access flaws.

Scripting allows to experiment and tune new analyses with ease, and extends the tool to the advantage of other users. The chosen script language, SQL, perfectly fits the need of accessing code structures and results of previous analyses. However once an analysis has been prototyped in SQL and tested, it can be encoded then in C for better performance.

Scripted analyses for MAQAO are inserted into menus with a simple configuration file. The analyses included in MAQAO can be sorted out in the following categories:

- *Simple statistics*: number of `nops`, number of bundles with multi-way branches or number of loops. This gathering can be executed on a part of the code structure (a particular function or some blocks), which would not have been possible with a simple `grep`. Moreover, using the optimistic cycle evaluation provided by the compiler for each instruction, it is possible to have a static performance evaluation of important loops.
- *Histograms*: histogram of basic block sizes in a function or histogram of the IPC in a function or block. The histogram of IPC shows the number of non-`nop` instructions scheduled at the same issue. As the compiler estimates the number of times each block is executed (using profiling), this helps to single out the lines of code where the IPC should be improved.
- *Code pattern detection*: pattern detection is a valuable tool in order to detect deficient sequences of code. Simple examples include the detection of missing prefetches in loops, or missing `fma`. For address computations, a pattern composed by `setf/getf` instructions indicates a conversion to floats, usually to perform a multiplication. More intricate patterns can be expressed, containing more than one instruction and with some interrelations with the underlying structure: for instance useless spill/fill sequences are sometimes generated by the compiler (no use of the variable between fill and spill, see Section 8.2 for an example). This spill/fill pattern can be either generated explicitly (with `ld.fill` and `st.spill` instructions) or implicitly (with a `ld/st` on the stack).
- *Compiler optimization detection*: MAQAO can find out if some optimizations were performed by the compiler. For instance, it detects pipelined loops and the parameters of the pipeline. For unrolled loops, hints can be given by MAQAO concerning the unroll factor, based on the match between source and assembly lines and the comparison of loads/stores in both codes. But due to the possible compiler optimizations, a reliable unroll factor is difficult to capture. Loop blocking, IPO and so on are not yet supported, but notice that for some codes, such as scientific ones, characterizing loop optimizations often boils down to the unrolling factor or pipeline depth.

Figure 2 shows two examples of script. Figure 2.a is used to detect the number of `setf` in the selected function. This instruction frequently appears in poor performance address computation. Figure 2.b computes the number of stalls in selected blocks according to the compiler. Indeed for each instruction, the compiler provides the cycle when it is issued. Comparing this value to the issue number gives the result. Each line of the table `blocks` has a column corresponding to the id of the including function, the same hierarchy applies for instructions and blocks.

Scripting new analysis involves the use of the database associated to the code. We provide some details concerning this database infrastructure: the instructions, bundles, blocks, directives, loops, functions and graphs are stored in 13 tables. The size of the database seems to depend linearly

```
select 'No of setf :'||count(*)
from blocks,instructions
where blocks.function=FUNCTION
and instructions.block=blocks.id
and instructions.name='setf';
```

a. Number of `setf`

```
select max(cycle)-min(cycle)-count(id)
from instructions
where bstop=1
and instructions.block in (SELECTED_BLOCS)
group by block;
```

b. Counting stalls in selected blocs

Figure 2. Two examples of SQL requests (`FUNCTION` is a macro substituted by the id of the current function).

on the size of the assembly code: for codes ranging from a few KB to 6 MB, the expansion factor remains between 4 and 5. The size of the database, especially the size of the instruction table, impacts the way queries must be written in order to be efficient.

To sum-up this section, MAQAO computes the structure of the assembly code and enables application specific analyses and expert knowledge to be integrated inside the tool, via new scripts. Code pattern detection can be combined with hotspot detection or other dynamic analyses. As a result, it has a great potential in revealing the possible flaws of the code generation.

5 Static Evaluation

In this section, a simple static performance model is presented. The goal of this model is to assess for simple vector loops (iterations being independent) the quality of the instruction schedule produced by the compiler. This model will be critical to quickly identify loops where potentially the compiler has poorly performed.

5.1 Static Performance model

In this section, we assume that loops are simple vector loops without dependencies between iterations. For each loop, MAQAO analyzes the assembly code generated and automatically sorts assembly instructions (except the `nops`) into various classes: Floating Point Add, Floating Point Multiply, Load Floating Point Instructions, Load Floating Pair Instructions, Integer Instructions, etc ... Each instruction class is then counted and assembled into instruction groups to compute simple issue bounds according to Itanium 2 specifications. The major instruction groups are: Floating Point Arithmetic Group (constituted of all of the Floating Point Arithmetic including Floating Point moves), Floating Point Memory Group (constituted of all of the Floating Point Load and Store Instructions including Prefetch Instructions), Integer Arithmetic Group (constituted of all of the integer arithmetic instructions), Integer Memory Group (constituted of all of the integer load and store instructions).

For each group, a simple bound based on issue limitations can be computed: for example if the Floating Point Arithmetic Group contains K instructions, since Itanium 2 can issue up to 2 Floating Point Arithmetic Instructions per cycle, then $K/2$ cycles are needed to issue these K instructions. This number is then called Floating Point Arithmetic Bound (FPAB). The Floating Point Memory Bound (FPMB) is more complex to evaluate because several rules have to be taken into account. Memory accesses are done through 4 memory ports (M0, M1, M2, M3) which can each service one request per cycle (resulting in a peak rate of memory accesses per cycle). While Floating Point Loads can be serviced by any of the 4 ports, stores can only be serviced by ports M2 and M3, prefetch by ports M0 and M1 etc ... Then all of these bounds are combined together to compute the Global Loop Bound (GLB) taking into account all of the limitations imposed by Itanium 2 architecture. GLB is fairly complex to compute, it amounts to solve a small Integer Linear Programming Problem. The Global Loop Bound corresponds to an ideal case assuming that there are no dependencies between instructions and that all of the operands are in L2. This bound is interesting to be compared with the number of issue cycles of the loop body (corresponding to the instruction schedule produced by the compiler).

Loop	FFTW2	FFTW4	FFFTW8
FP Fma	8	28	84
FP AB	4	14	42
FP Load	6	14	30
FP Store	4	8	16
FP MB	3	6	11
GLB	4	14	42
Issues	9	23	56

Table 4. Bounds for FFTW2 and FFTW4 codelets.

5.2 Example on FFTW

Table 4 gives the output produced by MAQAO on the FFTW2 and FFTW4 codelets. A simple look at the last two lines indicates an important mismatch between GLB and issue cycles. This indicates that the compiler has been constrained by dependencies between instructions. A look at the source code reveals array accesses (reads and writes) of the form $X[0]$, $X[ios]$ ... Since the compiler does not know the ios value, it performs a very conservative schedule resulting in poor performance. In fact the compiler is right to be conservative because if $ios = 0$, array access order has to be preserved. Unfortunately, for all practical cases, ios value is never equal to 0. MAQAO there was helpful to point at the instruction schedule problem. When refining the analysis, MAQAO indicated us that the load and store order was preserved while it was unnecessary due to the fact that ios value was never equal to 0.

MAQAO provides also additional information on loops such as the estimated total execution time in function and N the loop iteration count. For software pipelined loops, this information is extremely useful because combined with value profiling on loop trip counts, it allows to detect performance problems due time wasted in pipeline draining (see Section 8.1.2).

6 Dynamic Analysis

From a dynamic analysis side, MAQAO goes beyond mimicking other profiling tools: in addition to support hardware counters, and tracking where CPU cycles are spent, it performs value profiling. 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. Usually instrumentation for value profiling is done within the source code, but MAQAO does this at the assembly level. The goal is to prevent the compiler from changing code generation due to the presence of profiling probes (as a nice side effect this method induces very low run-time overhead). Therefore, doing the instrumentation after the compilation stage allows to observe the real application behavior with minimal disturbance.

Instrumentation Framework: instrumentation 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 in a dedicated memory zone some specific registers. MAQAO ensures that, by analyzing allocated registers or by adding spill/fill instructions, the register stack remained unchanged by the instrumentation.

Execution time profiling: from the GUI, probes can be automatically inserted around a selected code fragment. Code fragment size can vary from a function down to a basic block. An important aspect of this assembly probe mechanism is its ability to draw an accurate picture of the execution time: additionally to the total number of cycles spent in the function we get individual time for each function execution[1].

[1]Individual times are also used to detect if the ITC register was accessed in a burst mode (several times within 40 cycles) which degrades its accuracy from 6 to 40 cycles.

Value profiling: any register of high interest can be singled out for monitoring:

- *Function parameters*: distribution of parameter values for any given function is a clear indicator that code versioning is an optimization to consider;
- *Addresses used in load/store/prefetch instructions*: this allows to build the address stream of the first loop iterations. Then comparing the address patterns with similar address patterns tested via micro-benchmarking, alignment problems like bank conflicts or more subtle load store queue conflicts [1] can be detected.
- *Prefetch Analysis*: by tracking the target address of the prefetch instruction and comparing with the memory address used in load/store instructions (cf item above), prefetch distance can be computed and the arrays which are indeed prefetched can be determined.
- *ITC register*: direct access to this clock register allows a fine grain execution time profiling.
- *LC register*: used to store the number of iterations for every counted loops. This is a powerful parameter to evaluate relevance of software pipeline.

For hardware counters the implementation relies on standard tools for standard operations, which is similar to the scheme followed by `HPCview` [8]. Hence all the hardware counter management is basically done by interfacing *perfmon* [12]. Incorporating the analysis tree described by Levinthal [13] to help end-user to navigate among counter terminology would be a nice add-on.

7 Methodology for Guided Optimization

As presented in the previous sections, MAQAO provides a set of static and dynamic analyses of the compiled code. We propose thereafter a methodology, helping the expert user to locate performance bottlenecks and giving hints on how to remove these bottlenecks and optimize the code. We assume expert users are willing to either modify the source code, change compiler options (possibly resorting to black belt options), modify the assembly code or give hints to an optimizing scheduler such as XLG [22].

The step by step methodology is the following:

1. Determine most important functions and code sections by using a standard profiling tool.

2. Detect assembly code patterns of known poor performance. For each pattern detected, we associate some hints concerning possible optimizations, depending on the compiler used. For instance, `setf/getf` high latency patterns often results from poor performance address computation in Fortran codes. Porting the code from Fortran to C solves this issue because the address computation is better handled by the Intel C compiler.

Note that the database of detected patterns is evolving and can be enlarged through expertise or micro-benchmarking.

3. Once no inefficient assembly code patterns remains, we make an assessment on how far the code is from the optimal and what is the next bottleneck. For this, we evaluate three performance values:

   a. The static performance bounds presented in Section 5. They evaluate the density of computation and memory instructions in the code, and answer to the question 'how far is the code from optimal usage of functional units ?'. They are optimistic bounds: once reached, the code cannot be optimized further, as far as functional units are concerned.

   b. The number of instruction issues, according to the compiler model. This information is given through annotation in the assembly code.

   c. The results of cycle profiling.

   When the three bounds match (or are in a 10% interval), there is no need to optimize further. The other cases are handled by the next steps.

4. If there are any difference between bounds 3.b and 3.c, this comes from memory latencies. Comparing the results of an address value profiling (prefetches, loads and stores alike) with the memory access templates obtained from micro-benchmarking will determine the kind of bottleneck. Such a bottleneck can be a memory bank conflict, a wrong load/store queue aliases or missing or poorly parameterized prefetches. Versioning on the addresses solves the issue due to memory bank conflicts. Additionally black belt pragmas can be used to schedule differently load and stores so as to avoid aliases and they can be used to change prefetch distances. If any difference remains between bounds 3.b and 3.a, resort to classical performance counter methodology to identify the issue.

5. If the bounds 3.b and 3.a correspond, the compiler has an accurate performance model of the code. Detecting bubbles in the compiler static schedule hints for a higher factor of unroll. Indeed this enables better scheduling opportunities (with a jam for instance) and may remove the bubbles.

   Besides, on innermost loops, MAQAO can extrapolate in terms of cycle/iteration, the complexity of the loop. This evaluation is based on the issue count and the pipeline parameters (if the loop is pipelined) Making a value profiling of the loop trip counts and building a histogram of values can determine whether (or when) other versions of the code are more appropriate. As a matter of fact a deep pipelined loop is not appropriate for small loop trip count. Versioning the code at the source level, or preventing the compiler from software pipelining, solves this issue. On the reverse, it may be beneficial to pipeline a partially unrolled loop, provided the number of iterations is high enough. Compiling with '-O3' triggers software pipeline optimization, and removing false dependences (using `restrict` or `no-alias` compiler flags) may help the compiler. Other internal limits such as basic block size can prevent the compiler from performing this optimization. In this case, resorting to an optimizing scheduler like XLG [22] is necessary.

6. Value-profile the parameters of the hot functions (building histograms). If the same values often appear, it can be worth performing some function specialization. Recompile the versioned function and evaluate its performance with MAQAO.

After each code modification, the user can iterate the optimizing process. Up to now, these steps are not automatically processed, however most of the analysis/performance evaluations could be performed by MAQAO. This is still on-going work.

If there are remaining performance bottlenecks, then the expert has to resort to an optimization guided by performance counter analysis.

8 Case Studies

In this section we report results obtained using MAQAO to analyze and drive optimizations on two different applications. One being a traditional scientific code focused on floating point performance and stressing the memory bandwidth, while the second code is integer intensive with limited memory requirements.

Both codes were run on the same hardware platform: a BULL Novascale system populated with 256 Itanium 2. Each processor is running at 1300 MHz with 3MB of L3 cache. On the software side we use Intel C/Fortran compiler 8.1 [2].

8.1 TERA Benchmark

The first case study concerns the optimization of the *TERA* [14] reference benchmark. This benchmark, designed and used by the CEA-DAM (French atomic agency) in Fortran, consists in the resolution of fluid dynamic equations with precise methods. With a profiling phase, it appeared that two code sections deserve to be well optimized since they are the most time consuming sections of the whole benchmark. First, *Eis Loop* is a simple vector loop

[2]Intel Compiler C/Fortran v8.1.022, built on September 22nd, 2004

performing floating point intensive operations on several arrays whereas *Totalisation* involves a *while* structure with complex control flow driven by array values.

8.1.1 Eis Loop

The loop is defined as :

```
do i = first_cell, last_cell
  Eis(i) = T(i) - 0.5*(U(i)**2 + V(i)**2 + W(i)**2)
end do
```

We found that the best compiling options for this code are `-03, -fno-alias`.

Address Computations: Relying on simple pattern recognition (see Section 4.3), MAQAO finds that the compiler generated low performance code to compute addresses. Indeed on Itanium 2 architecture, Intel Fortran back-end uses high latency floating point instructions (such as `getf`, `setf`, `xma`) to compute integer addresses.
By experience, we know that this is a flaw of the Fortran compiler. Indeed by porting the code from Fortran to C, all array addresses computations are switched from floating point to arithmetic integer units.

Loop Fission: By inspecting the control flow graph and gathering line statistics from assembly code, MAQAO finds that from a single loop at the source level the compiler has generated two consecutive loops. The first of these loops is unrolled by a factor of 2 and software pipelined. The second loop is only software pipelined and served as epilogue code for the preceding unrolled loop. Due to the unrolling factor of 2, this means that the epilogue loop is at most executed once i.e. when trip count of the original loop is odd.

Therefore two sources of performance loss are isolated:

1. slow code generated for address computations,

2. cost of the pipeline prologue/epilogue.

As discussed earlier, the address computation problem is solved by porting the code into C.

The pipeline problem is handled by splitting the loop at the C level. Since the compiler performs unappropriated unrolling and epilogue generation, we hand-unroll 8 times the loop body and write a loop tail code for (at most) 7 additional iterations.

For the 8-unrolled version, the compiler generates a dense SWP loop (only 6% of `nop` operations) and makes an efficient usage of the large register file. Additionally, thanks to unrolling, a large number of loads are available allowing a schedule which prevents most of the load/store queue and bank conflicts. The loop tail code, displayed in Figure 3, is implemented in C as a `switch` section in which each

```
switch (remaining_iterations) {
  case 7:
    Eis[n+6]=T[n+6]-0.5*(U[n+6]*U[n+6]+V[n+6]*V[n+6]+
                         W[n+6]*W[n+6]);
  case 6:
    Eis[n+5]=T[n+5]-0.5*(U[n+5]*U[n+5]+V[n+5]*V[n+5]+
                         W[n+5]*W[n+5]);
  // ... and so on for case 5 to 2
  case 1:
    Eis[n  ]=T[n  ]-0.5*(U[n  ]*U[n  ]+V[n  ]*V[n  ]+
                         W[n  ]*W[n  ]);
}
```

Figure 3. Loop tail code for the improved version of Eis Loop.

`case` is not followed by a `break`. This allows a late entry into the *switch-case* structure.

For this structure, the compiler generates a fully predicated code (7 iterations). In this particular case this is far more efficient than the SWP version. Overall these two code transformations improve the performance of this loop by 22 %.

8.1.2 Totalisation Function

The other critical function of the benchmark is composed of nested loops with a complex control flow: a `while` structure with early exits and three vector loops.

Value Profiling: The code generated by the compiler is rather complex due to the highly conditional branching structure of this code section: the `while` loop containing early exits. MAQAO detects the use of software pipeline with unappropriated parameters for small vectors. Furthermore, this loop is preceded by a costly block of array address computations which in terms of execution time corresponds to 10 iterations of the main loop body. This is extremely high since MAQAO's value profiling analysis shows the loop trip count is constant and equal to 5.
The appropriate solution is to specialize this loop for 5 iterations. After converting the code from Fortran to C (to avoid address computation, see section 8.1.1), versioning is applied. The peeled case (5 iterations) is fully unrolled in C. For this code the compiler generates a predicated code, giving better performance than the software pipelined version. Of course, for other trip count values, original loop is kept back and a software pipelined loop is generated. The versioning technique adds several arcs in the control flow graph to select at runtime which is the correct version to execute but overall this new function outperforms the original one by 20 %.

8.2 SHA-0 Attack

The second case study concerns the optimization of a cryptanalysis application: the SHA-0 attack. This algorithm was developed by Chabaud and Joux [15] and this implementation was the first to find a full collision on SHA-0 in August 2004 [16]. We used the best found compiling options: `-O2 -fno_alias`.

Program Overview : Written in C, the attack code requires high performance integer computation. Indeed the algorithm manipulates integer values on 32 bits (because SHA-0 was designed for 32 bits architectures). The second characteristic of this code is complex control flow. For each pair of messages, the algorithm applies SHA-0 encryption turn after turn (out of 80 turns) in parallel. But during these encryptions, the program contains *early exits* to stop the computation on these 2 messages when it is sure that they will not collide at the end of the 80 turns. Because of these *early exits*, the control flow is very complex and unpredictable.

We investigate the most time consuming function, called `do_neutral`, that takes about 70 % of the whole CPU time. `do_neutral` compares messages belonging to the same set called *neutral set* (i.e. having in common several properties on their bits). Iterating on this set allows to reuse some computations made for previous messages.

Register file pressure: MAQAO static analysis highlights a potential bottleneck due to the high pressure on integer registers. The compiler generates many implicit *spill/fill* instructions (found by pattern recognition) pointing out that, at some points, more than 128 registers are alive. But analyzing the assembly code, some of these instructions are superfluous. Indeed, in the code displayed

```
{.mii
 st8 [r31]=r67
 add r31=176,sp //r31 is now set to address sp+176
 nop.i   0 ;;
}
{.mii
 ld8 r8=[r31]   //load value at sp+176 in r8
 add r31=176,sp //r31 is re-set to the same value sp+176
 nop.i   0 ;;
}
{.mii
 st8 [r31]=r8   //r8 is stored at the same address (sp+176)
 add r31=184,sp
 nop.i   0 ;;
}
```

Figure 4. Simple example of implicit *spill/fill*

in Figure 4, it is obvious that the last store saves an unmodified value (`r8`) loaded a few instructions before. So the two additions, the load and the store associated to `r8`, can be safely removed.
The same pattern appears across basic blocks suggesting potential problem in data flow estimation by the compiler.

Counter	Value
Cycles CPU	24 309 980
Stalls Cycles	3 266 458
Stalls Cycles due to L1D	1 173 888
L1D Misses	76 099

Table 5. Hardware counters on `do_neutral` function. Notice that a large amount of stall cycles is due to L1 Data cache misses.

Replacing theses instructions with `nop`s or deleting them (with rescheduling) shows that they are useless. Even if their removal does not bring a significant speedup, it is more than plain dead code elimination since these instructions were executed and memory requests were issued. A more ambitious scheme would use these freed slots as hoisting opportunities. Another scheme under investigation is to use memory fences to split `do_neutral` in smaller subfunctions to facilitate register allocation for the compiler.

Memory/Cache Interference: Because SHA-0 algorithm uses exclusively 32 bits integers, the SHA-0 attack intensely uses 32 bits integers too. On the generated code this is showed by many memory interactions on 32 bits. By micro-benchmarking training (see Section 4.3), MAQAO knows that 4B interactions lead to performance loss on Itanium 2 architecture. This is mainly due to the potential 8B unalignement. Some banking structure of 8-Byte width appears on the data path for store to the L1D cache (see [19] page 60). Indeed in a 32 bits arrays, one cell out of two is not aligned on a 64 bits boundary. Furthermore, on dynamic side, table 5 displays the list of hardware counters related to cache/memory for one call of `do_neutral`. Hardware counters confirm the high number of memory interactions. It seems that stalls occur due to the excessive pressure on L1D [3]. This is surprising, since the overall memory footprint is around 5 KB, fitting perfectly in L1D.

Taking into account these two information, we use padding to align load/store on 64 bits boundary. Even if padding is a well known optimization to reduce cache conflicts, in our case, we used it to align each array cell. As a side effect, memory footprint doubles (because each cell size grows from 4B to 8B) but, applying this optimization on a part of `do_neutral` function we obtain a speed-up of 2 (execution time is halved !) with the same compiling options (see Table 6). As an unexpected conclusion, increasing the memory footprint to align data on 64 bits boundary increases the performance by a factor 2 avoiding almost all bubbles due to L1D access. Notice that the

[3]Measurement is based on BE_L1D_FPU_BUBBLE_L1D perfmon counter.

Counter	Original	Optimized (padding)
Cycles CPU	763.1	368.8
Numbers of instructions	1245	1326
Stalls Cycles	498.6	63.3
Stalls Cycles due to L1D	118.4	3.0
L1D Misses	57.2	1.0
Number of loads	278	270
Number of stores	159	154

Table 6. Hardware counters on part of `do_neutral` function: original version and optimized with padding.

number of instructions retired increases with padding optimization because, instead of accessing directly the arrays, we need to multiply indices by 2.

9 Conclusion

As shown in the case studies MAQAO already fulfills needs as a convenient and powerful analysis platform. It allows to quickly detect problems in the assembly code generated and it gives useful hints on fixes.

Addressing the performance problem at the assembly level seems relevant, especially on Itanium 2 due to the EPIC instruction set and the importance of compiled code quality. The ability to navigate and present the restructured code in its hierarchy allows end-user to build up incrementally his knowledge base. SQL offers a flexible interface to dive into the code structure tracking potential performance problems in particular known poor performance code patterns. The combined use of a static model and value profiling capabilities allows to refine quickly code performance analysis and offers capabilities complementary to the standard hardware performance counter based tools.

Finally, based on MAQAO, the performance analysis methodology proposed allows to perform efficient code optimization.

Future work include automating the process of performance detection and optimization lending towards an assembly to assembly code optimizer. This would include for example deletion of useless spill/fill instructions, and load/store instructions rescheduling.

References

[1] Christophe Lemuet, William Jalby and Sid Ahmed Ali Touati. Improving Load/Store Queues Usage in Scientific Computing. ICPP 2004: 38-45

[2] Christophe Alias and Denis Barthou. On the Recognition of Algorithm Templates. Electr. Notes Theor. Comput. Sci. 82(2): 2003

[3] W. Jalby and C. Lemuet. Exploring and optimizing Itanium2 cache(s) performance for scientific computing, 2nd Workshop on EPIC Compilers and Architectures, held in conjunction with MICRO35, November 2002. Istanbul, Turkey

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

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

[6] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. Graph Drawing 2001: 483-484

[7] 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

[8] 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

[9] 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

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

[11] 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

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

[13] David Levinthal. Building and Optimizing Applications for the Intel(R) Itanium(R) Processor, ClusterWorld Conference, San Jose, CA, May 2004. http://www.clusterworld.com/CWCE2004

[14] CEA DAM, French Atomic Commission. TERA project review (in french), CHOCS : revue scientifique et technique de la Direction des Applications Militaires, Num. 28, October 2003

[15] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0, CRYPTO '98: Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology, 1998, pages = 56–71, Springer-Verlag,

[16] E. Biham and R. Chen and A. Joux and P. Carribault and W. Jalby and C. Lemuet. Collisions of SHA-0 and Reduced SHA-1, EUROCRYPT'05 2005

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

[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] Intel Corporation, Intel Itanium 2 Processor Reference Manual For Software Development and Optimization, 251110-002, April 2003

[20] Intel Corporation, Introduction to Microarchitectural Optimization for Itanium 2 Processors, 251464-001, 2002

[21] Intel Corporation, Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference rev. 2.1, October 2002

[22] Caps Entreprise, www.caps-entreprise.com

[23] 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

[24] 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

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