MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2

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

Contact: William.Jalby@prism.uvsq.fr

[1] LRC ITACA CEA/DAM - Université de Versailles Saint-Quentin,
[2] Prism, Université de Versailles Saint-Quentin,
[3] Bull Corporation

Quality of the code produced by compilers is essential to get high performance. Therefore, being able to assess precisely code quality is extremely important. This issue can be successfully tackled by using performance counters and dynamic profiling. In this paper, we advocate that in many interesting cases, a careful static analysis of assembly code can achieve similar results at a much lower cost and with a better accuracy. The principles of an automatic tool (MAQAO) for performing such an analysis are presented. Among its key advantages, MAQAO offers versatility (the user can specify a particular analysis using SQL formalism) and precise diagnosis capability which can be later used for carefully driving the optimization process. Two case studies on real codes are presented to illustrate the power of the tool: in each case, MAQAO helped us locate performance problems easily and define an optimization strategy leading to substantial code improvements (20 to 30% on the overall appliaction execution time).

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 Nowadays, with the recent generation microprocessors, such simple metrics are no longer valid. First caches have introduced data locality as a key metric for performance. Then, more other metrics have to be taken into account such as number of branches, use of specific instructions (such as fused multiply add, predicated instructions, management of memory hierarchy through prefetch etc). Additionally, some low level interactions such as bank conflicts or load/store queue wrong aliasing require to analyze in detail memory address stream. In fact, on modern microprocessors, taking into account all of the architectural features is critical to get high performance: for example on Itanium systems, *gcc* which performs most of the standard (architectural neutral) optimizations is very often outperformed by Intel C Compiler (*icc*) because *icc* is taking into account all of the features offered by the Itanium 2 [19, 20]. However this performance gain comes at the expense of using complex optimizations such as multiversioning: i.e. for the same code fragment, several versions are generated which will be dynamically selected at run time depending on parameters such as loop iteration count. As a consequence, first as usual complex optimizations generate code hard to analyze and additionally with potentially unstable performance (for example due to bad choice for the strategy selecting the different versions).

On the other hand, performance analysis has made tremendous progress with the appearance of various low level hardware capable of tracking various events. Such counters are extremely helpful to locate performance bottlenecks: for example bad data locality automatically generates high cache miss ratio which can be easily captured by such counters. In a similar manner, branch prediction or code sparsity (average number of nops per bundle) can be analyzed. In all of the cases just described, performance counters even allow to correlate performance problems and source code line simplifying performance optimization [1]. Unfortunately, such ideal cases are not so common rule.

For example, missing an opportunity of using an `fma` instruction can be much trickier to detect if not impossible. If the code is as simple as a DAXPY, using performance counters to evaluate the number of `fma` will allow to detect that the compiler does not use `fma`. Now, if the loop is more complex, containing several opportunities for using `fma` and that the compiler missed some of them, counting `fma` will not be enough. In such a case, inspecting directly assembly code will be much simpler to reveal the problem. A similar situation will occur with prefetch instructions: for a loop, the compiler can have inserted prefetch for some array access and forgotten to insert prefetch for others [2]. Again, a simple inspection of the assembly code will be much more efficient to point out directly to the right source of the problem. Also all of the cases where a compiler missed "classical" optimizations such as constant propagation, common subexpression can be detected much more easily on the assembly listing than by hardware performance counters. A few more examples are given in the "Case Study" section.

Now the real problem is "inspection/analysis" of assembly code: the first goal of our MAQAO tool is to automate as much as much possible and in a flexible manner, assembly code analysis. A synoptic view of MAQAO tool is given in figure 1.

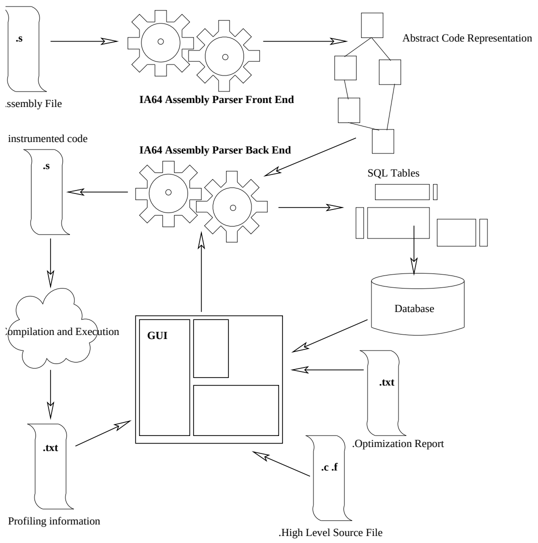


Figure 1: Overview of the assembly tool MAQAO.

The first step takes as input assembly files and parses them to produce a structured representation of the assembly code: structured representation means building call flow graph, control flow graph, loop structures. From that representation, SQL tables are built and organized in a database. Now on this database, queries such as detecting `fma` use can be performed, mimicking the manual search through the assembly file. Having structured the assembly code is not only usefull for queries but also for modifying it. A first application

[1] It should be noted that this correlation is very often achieved through sampling which in some cases can strongly bias the results and alter their validity

[2] More subtle and harder is the case where the compiler did insert prefetch but the prefetch distance is too short !

of this ability is insertion of performance probes (instrumentation) directly at the assembly level. Other applications are direct modifications of the assembly code to improve code quality, perform optimizations.

Now, for assessing code quality, detecting bad code sequences is not enough, we need to build some "reference" metric: i.e. we need some way of evaluating what an "optimal" compiler should have done. More precisely, we compare performance metrics computed on the generated code with "optimal" bounds. For that, we used a performance model introduced by E.S. Davidson (MACS) which provides simple performance bounds which are usefull for quantifying the quality of the code produced.

Section 2 gives an overview of various performance tools which are tackling code analysis and optimization. Section 3 describes the MAQAO static analysis phase, how the asssembly code is structured and how pattern detection can be performed. Section 4 details how to compute "optimal" performance bounds and how to compare them with the code produced by the compiler. Section 5 presents code instrumentation at the assembly level. Section 6 illustrates how the diagnostics produced by MAQAO helped us optimize two real life codes: a scientific code called TERA, and SHA-0 attack, a cryptographic application. Finally, Section 7 very briefly indicates how MAQAO can be used for interfacing and driving optimizations. Finally, conclusion and openings are provided in section 8.

2 Related Works

Two main classes of Low Level instrumentation tools can be related to MAQAO. One calss 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-dependent). The other family of tools is more focused on code 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 `Finesse` [9], this later one being more focused on parallelization than code optimization.

2.1 `VTune`

`VTune` [5] is the well known Intel promoted performance evaluation tool. Basically it provides classical profiling information as well as a link to hardware event monitors, a view of the corresponding assembly and high level code. Additionally, all the features are embedded within an elegant GUI. Nevertheless, among its weakness, `VTune` relies on sampling, and despite complex heuristic to set the sampling rate, the tool remains more convenient to detect hotspot than really evaluate the whole assembly quality. Furthermore, while `VTune` is very good at profiling and hotspot detection it falls short on code instrumentation.

2.2 `PAPI`

The *Performance Application Programing Interface* [11] (`PAPI`) is an interesting initiative to abstract, through the usage of a standardized API, the too often vendor specific hardware monitors. `PAPI` only selects a subset of metrics as being meaningful which may be the right way to proceed in front of the numerous counter available. Still, for tools writers who see abstraction as a burden, `PAPI` also supports a native mode, i.e. a direct access to the proprietary hardware monitors.

An open question for MAQAO is to be compliant with `PAPI` or not: since the component architecture of MAQAO introduces a relative independence between module, using `PAPI` seems to be the choice of consistency. However, calls to `PAPI` needs to be done at the high level source code. And we already develop a low overhead assembly instrumentation scheme which is very promising, we should stick with it as much as possible.

In our sense `PAPI` should be kept as an inspirational source for the development of performance metrics, or in case of migration from Itanium to a new platform.

2.3 Salto

Salto [7] is a framework dedicated to the implementation of complex manipulations of low-level codes. One of the developer's main goal is to provide an object-based user interface to implement these transformations directly on the abstract representation of assembly programs. Therefore, users can concentrate on the implementation of actual optimizations and code instrumentation methods, without worrying about syntax and lower level details handled directly by Salto.

The main difference with our approach: Salto is a toolkit not a tool, everything has to be explicitly coded and it does not include any optimization knowledge like MAQAO tool (even if it allows to express it via programming Salto applications). It does not provide database accesses and requests to gather statistics, end user has to hard-coded every information he is interested in. More oriented toward code re-ordering and transformation than observation and instrumentation, even if it clearly offers a super-set of our own instrumentation routines (like catching the loop trip count), it lacks support for hardware monitor access and the prototyping flexibility brought by SQL requests.

Basically Salto appears as a back-end of MAQAO diagnostic chain: the problem is identified and some transformations has to be applied to solve it.

2.4 ATOM and Pin

Both tools offer similar features, the main difference being that Pin [10] is targeted for Intel processors (x86, IA64, IA32-64 and even Xscale) codes. Additionally while ATOM [4] is focused on Alpha assembly and that Pin allows to instrument not only assembly files but also binary files even already loaded in memory (therefore supporting dynamic instrumentation). Basically the idea is to branch on user defined instrumentation routines when specific instructions are executed. For instance, all branches within a code will be preceded by a call to a routine which can be as simple as incremented an branch counter. At the end of the execution the user will get the exact number of branches executed in the code. More ambitious instrumentation can be executed, such as trapping all load/store of the code and simulated the effect of larger or deeper cache hierarchy. Using ATOM or Pin such task can be easily tackled, user needs to write the cache simulator and keep track of the cpu cycle.

ATOM has enjoyed a large success within the research community, leading to the development of execution driven simulators. Such simulators are much more accurate than the previous generation of trace driven tools [3]. Now, that the original platform is fading away, Pin is promised to a similar success within the Itanium community.

However, while being very useful these tools are more oriented toward prospective architecture simulation than code performance analysis, they lack the support of the now commonly available hardware performance counters and are not suitable to evaluate code quality.

2.5 HPCview

HPCview [8] is currently the main competitor for MAQAO tools, firstly it is addressing the same problem: the complex interaction between source code, assembly, performance and hardware monitors. Secondly its development is much more advanced and they have already reach a production level.

Basically HPCview is providing a well designed GUI based on web browser, providing simultaneous view of the source code, assembly code and dynamic information. This interface is connected to a database for each statement of the assembly code, with the summary of the dynamic information related to it. Furthermore based on control flow graph and a tool named bloop, HPCview provides an abstracted view of code loop structures (using an XML interface).

However, some important difference should be underscored:

- heterogeneous application: XML, HTML, bloop, all of this leads to a complex software environment to maintain.
- While a database is embedded in the application, end-user as only limited opportunity to explore the code.

- limited set of performance metrics: HPCview uses miss ratio, execution time and a very limited set of metrics to help the developer to optimize its application
- lack of data analysis, loop trip count or other information collected dynamically can leads to powerful yet simple to implement optimization, such thing is neglected by HPCview.

Therefore the existence of this tool can be interpreted as a confirmation of the relevance of our approach, it is also a nice goal to meet for the quality of its GUI but still it is too much based on commodity available information instead of building more meaningful metric about the 'why' of the performance behavior.

3 Static Analysis

MAQAO parses any assembly code produced by *icc* and then performs several static analyses on the code. In addition, this is an important feature of the tool, the user has the possibility to widen the range of analyses and to do some fast prototyping of new analyses with a scripting module. From the parsing of the code to the detection of the loops, all the results of static analyses are stored into a database associated to the program analyzed. We detail in this section the main features of these phases.

3.1 Parsing Assembly

The instruction set reference of the Software Developer's Manual [21] for Itanium describes in more than 230 pages the IA64 instructions. Hand-coding a parser for such an important amount of instructions is error-prone and very time consuming. So instead of doing it by hand, we designed a script that parsed automatically the manual in pdf form. This script generated the assembly code parser used by MAQAO . Some simplifications have been performed on the code generated, additional rules for the assembly directives have been incorporated and the lexical analyzer have been built then by hand. The overall cost of developing the front-end of MAQAO has been light and it ensures that every IA64 instructions can be handled correctly. Note that this process could be easily adapted to other architectures.

Instructions and assembly directives alike are stored in the database. 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 through comment lines some values concerning the execution latencies (for each instruction issue), the execution number of 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 for static or dynamic analyses.

3.2 Computing the Structure of the Assembly

We now describe the different analyses forming the foundations for more complex analyses. These analyses restructure the assembly code in a hierarchical way, from functions to basic blocks and bundles. They provide a convenient structure for the user to browse through the interface. This interface consists in one window displaying the assembly and source codes, the call graph and control flow graph of a selected function, and has a editor for scripts.

The hierarchy of structures is the following, from the topmost to the lowest level:

- Call graph: this structure slices the code into different functions and shows how they interact one with the other. From a given function, the number of calls to a 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 the basic blocks inside a function, showing the possible control paths. This information is important for the detection of loops and provides a quick assessment of the complexity of the function. Selecting a node highlights the block selected in the

assembly code and positions the source code to the corresponding line. Likewise, selecting assembly or source code selects the nodes corresponding to the basic blocks associated to these lines;

- Loops: the loops correspond to the basic blocks that are likely to be executed the most, hence are a legitimate focus for analysis and optimization. They are stored in the database as list of blocks, with one particular block for its entry. Moreover, the domination tree of each function is stored as well: Each block in the database corresponds to an interval of integers. One interval of a block is strictly included in another if and only if the first block is dominated by the latest. This representation encodes the transitive closure of the dominance relation, that would be otherwise difficult to express in SQL;
- 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 instead if necessary.

An implementation of the reaching definition analysis is in progress; it will enable a dependence representation of the program.

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.

3.3 Scripting More Analyses

One of the key features of MAQAO is the possibility for the user to script new analyses with the SQL language. This offers him a wider range of analyses when standard statistics are not enough. The knowledge base of interesting analyses is built from the experience of different users. Scripting allows them to experiment and tune new analyses with ease, and extends the tool to the advantage of other users. The script language chosen is SQL, allowing the user a complete access to the structures of the code and to the results of previous analyses. Once an analysis has been fast prototyped in SQL and tested, it can be encoded then in C for better performance.

Scripted analyses provided with MAQAO can be inserted into menus with a simple configuration file. The analyses included in MAQAO can be sorted into these categories:

- Gathering of simple statistics: number of `nops`, number of bundles with three-way branching or number of loops. This gathering can be performed on a part of the structure of the code (a particular function or some blocks), which would not have been possible with a simple `grep`. Moreover, using the optimistic cycle evaluation given by the compiler for each instruction in a basic block, it is possible to compose them with profiling information in order to have an static optimistic performance evaluation of important loops for instance (the number of executions of basic blocks is indicated in the assembly through comment lines for profiled compilations);

- Histograms: histogram of basic blocks size 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 can give the number of times each block is executed (using profiling), this can help in showing the lines of code where the IPC for instance should be improved;

- Code pattern detection: pattern detection is a valuable tool in order to detect deficient sequences or code patterns. Simple examples include the detection of missing prefetches in loops, or missing `fma`. For the computation of addresses, the pattern composed by `setf/getf` instructions indicates a conversion to floats, usually to perform a multiplication. This poor pattern of address computation can be tracked down in the structure of the code. It is also possible to express more intricate patterns, including more than one instruction and with some interrelation with the underlying structure: spill/fill sequences are sometimes generated by the compiler uselessly (no use of the variable between fill and spill, see Section 6.2 for an example). The spill/fill pattern can be either generated explicitly (with `ld.fill` and `st.spill` instructions) or implicitly (with a `ld/st` on the stack). Detecting a useless spill/fill pattern can be expressed in SQL;

- Compiler optimization detection: MAQAO can find out some optimizations performed by the compiler. For instance, it detects pipelined loops and the parameters of the pipeline. For unrolled loops, hints can

```
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 'Size: '||size||' Number: '||count(*)
from (select blocks.name,
count(*) as size
from blocks,instructions
where instructions.block=blocks.id
and blocks.function=FUNCTION
group by blocks.name order by size
) group by size;
```

b. Histogram of block size

Figure 2: Two examples of SQL requests.

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 optimization, a reliable value of unroll factor is difficult to capture. The optimization report gives this information and could be incorporated if needed.

Figure 2 shows two examples of scripts. Figure 2.a is used to detect the number of `setf` in the selected function, usually showing a poor address computation, and Figure 2.b generates the histogram of block sizes. In both scripts, `FUNCTION` is a predefined macro substituted by the id of the current function. 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 analyses 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 on the size of the assembly code: for codes ranging from a few KB to 6 MB, the factor of expansion remains between 4 and 5. The size of the database, especially the size of the instruction table, has some influence on the way the queries must be written in order to be efficient. Though, as an interpreter for the SQL scripts is embedded in MAQAO , a "test/fail" approach can avoid the performance issues that may arise.

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, with new scripts. Code pattern detection can be combined with the hot spot detection or other dynamic analyses. As a result, it has a great potential in revealing the possible flaws of the code generation.

4 Static/Dynamic Performance Evaluation

Performance evaluation techniques are critical for the design of processor architecture as well as software development. Numerous researches are focused on the gap between theoretical and delivered performance and on the key optimizations to apply.

To predict performance two approaches have been proposed: (i) Static approach, taking into account factor evaluated statically within application (ii) Static/Dynamic approach, feeding the static analysis with data gathered dynamically. Dynamic analysis allows to obtain value undecidable by a pure static scheme.

Davidson et al.[13], [14] has proposed a performance model, called MACS bounds, taking into account both application and architecture specific parameters. Authors's analysis on source and assembly codes provides a series of performance bounds that explicitly identify the deliverable performance of the application and the individual contributions of several factors to the performance degradation. In this section, we propose an adaptation of this model for Itanium 2. In the model, cache effects, I/O latencies and interrupts are ignored.

4.1 The MACS Bound Model for Itanium 2

MACS bounds offer a model for peak performance in a hierarchical manner, each level designated by a name: M for the peak performance of the machine independently of the application, MA for the performance of the application, independently of the compiler, MAC for the assembly code independently of its schedule and MACS for the real code.

We focus the performance study on three kinds of essential operations: memory operations, floating-point operations and branches. These essential operations correspond to dedicated functional units in Itanium 2 and the aim of this performance is to assess the efficiency of the code w.r.t the maximal capacity of these units. Integer operations are not counted since they can be handled by different types of functional units (memory or integer). Performance is measured as cycles per floating-point (CPF) operations, cycles per memory access (CPM) operations and cycles per branch (CPB) operation.

We assume that profiling provides the number of loop iterations in the code examined or that a single iteration is analyzed.

4.1.1 MA and MAC Bounds

For the MA bounds, we count the minimum number of operations needed in the high level code, for each essential operation type. Each floating-point multiply, add, divide, comparison, min/max operation counts as one floating-point operation. Each loop, if, switch and break statement counts as one branch. And each array access counts as one memory operation. The sum of these counts for an application defines resp. the number of floating-point operations, denoted t_f, the number of branch operations, denoted t_b and the number of memory operations, denoted t_m. The Itanium 2 architecture constraints sequences of three operations to fit into bundles (or two in one case). As only some combinations of operation types are authorized in a bundle, we look for the number of authorized bundles that contain the t_f floating-point operations, t_m memory operations and t_b branch operations.

Finding the minimum of bundles fulfilling this condition corresponds to finding a solution to the following system:

$$\begin{aligned}
t_f &= x_{mfi} + x_{mmf} + x_{mfb} \\
t_m &= x_{mii} + 2x_{mmi} + x_{mfi} + 2x_{mmf} + x_{mib} + x_{mbb} + 2x_{mmb} + x_{mfb} \\
t_b &= x_{mib} + 2x_{mbb} + x_{mmb} + x_{mfb} + 3x_{bbb},
\end{aligned}$$

where the integer positive variable x_{abc} denotes the number of bundles of type abc used for the program, a, b and c are among $mfib$. Finding the minimum value of:

$$c_{MA} = \left\lceil \frac{\sum_{\text{authorized } abc} x_{abc}}{2} \right\rceil,$$

gives the minimum number of cycles necessary to execute the $t_f + t_m + t_b$ essential operations. Indeed, on Itanium 2, at most two bundles can be executed per cycle.

The MA-performance bounds are then computed in CPF, CPB or CPM with: $MABound(CPF) = c_{MA}/t_f$, $MABound(CPB) = c_{MA}/t_b$ and $MABound(CPM) = c_{MA}/t_m$.

For the MAC-performance bounds, the essential operations are counted this time on the assembly code: `nop` instructions are not counted as essential operations; Counted floating point operations correspond to arithmetic computations on floats (conversions are not counted), counted memory operations correspond to load and stores (fetch or integer arithmetic operations in m slots are not counted) and counted branches correspond to `br`. The same model as MA then applies similarly.

The types of two successive bundles are constrained so that they can be executed in one issue. A more complex model taking into account this constraint consists in counting couples of successive bundles with some variables. So for instance, instead of counting the bundles `mmi` with x_{mmi}, we could count the number of bundles `mmi` followed by `mbb` with a variable $x_{mmi|mbb}$. According to the microarchitecture manual, 58 variables are then necessary.

4.1.2 MACS Bound

At this level, the performance model takes into account the schedule of the assembly code. *icc* generates through comments the issue slot for each instruction. Therefore, c_{MACS} is directly taken from these annotations in MAQAO and t_f, t_b and t_m are computed in the same manner as for the MAC Bound. Finally, $MACSBound(CPF) = c_{MACS}/t_f$, $MACSBound(CPB) = c_{MACS}/t_b$, $MACSBound(CPM) = c_{MACS}/t_m$.

4.2 Gaps Between Bounds

The MACS Bounds hierarchy gives some indications on the origin of performance degradations by examining the gap between two successive hierarchy levels. We now detail the explanation for each kind of gap. Note that performance is expected to decrease from M, MA, MAC, MACS down to the measured performance according to the previous model.

The gap between M and MA bounds (GAP A) shows the dominant essential operations in the application: the narrower the gap for one kind of operations, the more important this kind of operation appears in the performances of the application.

The gap between MA and MAC (GAP C) shows the quality of the code generation and the optimizations performed by the compiler (apart from the scheduling). In some cases, the number of essential operations may vary between MA and MAC levels: predicating if statements or loop unrolling lowers the number of branch operations. Likewise, scalar promoting arrays lowers the number of memory accesses and the use of double load operations can drastically decrease this number. These factors may improve the performance of MAC w.r.t. MA.

The gap between MAC and MACS (GAP S) provides some hints about the quality of the schedule. It also reflects dependence constraints that are not taken into account on the MAC level and shows the impact of operations that are not considered as essentials (prefetches, address computations,...).

Finally, the gap between MACS and the measured performance (GAP P) shows the effects of all the interferences that have not been put into the model (memory hierarchy, I/O latencies, memory bank latencies,. . .). This also corresponds to what have not been taken into account by the model integrated in the compiler, as we are relying on its performance model for the MACS level.

4.3 Example: DAXPY

We study the case of a DAXPY code from BLAS1, performing the computation $Y = \alpha.X + Y$ on the vectors X and Y. The loop have been unrolled 16 times with some compiler specific options.

Figure 3 presents the bounds for the MACS hierarchy. t_f, t_b and t_m represent the number of essential operations in one iteration of the code. There are 16 `fma` in MAC since the loop is unrolled. The c row indicates the number of cycles necessary to execute the essential operations, with the constraints on bundles. For MA, a solution minimizing the number of issues is to consider two bundles: `mmf` and `mfb`, thus taking one cycle per iteration. For MAC, one of the solution is a pattern of 16 bundles `mmf` and one `mib` for instance, thus taking 9 cycles. The code generated by the compiler takes 12 cycles/iteration.

The performance is indicated by the row CPF, CPB and CPM. Only these values are given for the M level and for the measured performance, since others are code-dependent. GAP A shows that the preponderant operations are first floating point operations and then memory operations. GAP C highlights the fact that the compiler outperforms the CPM for the MA model: it uses double loads instead of simple ones. For CPF, the degradation comes from the increase of issues/iterations and CPB gap results from the unrolling.

GAP S for this simple example outlines the importance (for CPF and CPM) of the non essential operations: there remains in the code some explicit additions for the computation of addresses, there are 3 nops and 3 fetches. The measured cycle performance has been obtained through the dynamic analysis of MAQAO (see next section for details) on a DAXPY with 100000 elements. The value of 13.5 cycles is the mean value of several executions. The gap between MACS and the measured performances results from cache misses.

	M	MA	MAC	$MACS$	measured
t_f	n/a	2	16	16	n/a
t_b	n/a	1	1	1	n/a
t_m	n/a	3	32	32	n/a
c	n/a	1	9	12	13.5
CPF	0.5	0.5	0.56	0.75	0.84
CPB	0.33	1	9	12	13.5
CPM	0.25	0.33	0.28	0.37	0.42

Figure 3: MACS Bounds for DAXPY

The MACS bounds model provides an additional measure for the application. The sizes of the gaps provide very useful information to help the user identify the bottlenecks. The gaps between the different bounds give insights about the possible performance loss resulting from the code generation, code scheduling or cache misses. In future work, this model could be extended to take into account the effect of cache misses.

5 Dynamic Analysis

Profilers like `gprof` or `tcov` are the main tools used by performance conscious developers. Furthermore with the always increasing exhaustiveness of hardware counters on all modern processors, illustrated by `PAPI` [11], dynamic data collectors such as `VTune` [5], `cprof` have proved to be highly useful to understand the behavior of an application and identify bottlenecks. Therefore, as a platform for code analysis and optimization it was required for MAQAO to provide at least equivalent services.

But MAQAO goes beyond mimicking other tools: additionally to support hardware counters, and execution time profiling, it is also performing value profiling to monitor key parameters such as loop trip count, functions arguments, array offset and other performance sensitive data.

At the end of the execution the user has access to these dynamic values, as a summary but also in detail, individual values, distribution of trip counts for a particular loop, or standard deviation of execution time for a given function. This extra information leads to a better understanding of the application and highlights potential optimization tracks as illustrated in section 6.1.

5.1 Instrumentation level

Usually code instrumentation is done at the high level, performance probes being inserted within the source code. But this method introduces a subtle yet dramatic observation bias. By adding function calls within the source code it is changing the way the compiler applies its optimizations. As a result once instrumented, an application can behave in a largely different way: for instance, with degraded loop unrolling, branch optimization or inlining policy.

Instrumenting at the high level is simply not the right place to do it. Therefore MAQAO addresses the problem at the low level: assembly code.

5.2 Instrumentation Framework

While the static analysis part of MAQAO is standalone, for dynamic instrumentation we need to access all assembly sources and libraries used to build a binary of the application. We think this is a very light constraint since even a disassembled binary can be re-built without the compiler just using an assembler (*ias* for instance). Currently we do not handle cross-compilation (or deported compilation/execution) and everything as to be done either directly on the target system or implies some files copying from the user.

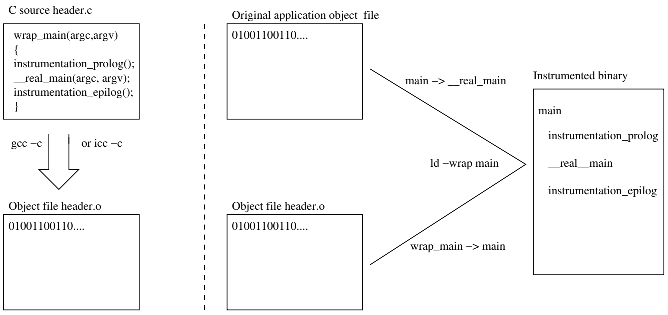


Figure 4: Overview of the wrapper: `main` from the original code is substituted with header `main` which performs memory allocation for statistics gathering and call a function to build the summary of these collected data.

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 through *perfmon* [12], while more specific tasks are handled natively by the tool itself.

The instrumentation is done first by encapsulating the `main` of the application (depicted in Figure 4), then injecting a limited number of extra-bundles, named assembly nodes, around the targeted code fragments to monitor. These bundles are in charge of storing in memory some specific registers.

Part of the problem is to guarantee that registers stack is unchanged after instrumentation. Either some registers are available or spill/fill action is required to reconstitute identical register stack. In a similar way, any added function call is surrounded by special registers save/restore code.

Our instrumentation allows to track specific values: loop count, clock, or even a general register, with a very limited overhead. These results are stored in buffers all along the application execution time. In case of buffer overflow [3] a special C function is called. This function builds up summaries and computes statistics before reseting the buffer.

5.3 Profiling supports

Traditional profiling is done through MAQAO interface if the source code is available by compiling the application with the classic profiling flags executing the corresponding binary.

If the source code is not available, to be consistent with our assembly focused approach, our tool supports an 'all functions' instrumentation which injects few bundles of assembly code to get and store the value of the clock register (ITC) at the entrance and exit of every functions. This allows to re-construct a complete profile of the application. However it does not evaluate the amount of time spent in library functions.

Through the Call Graph displayed in the GUI, user can select for instrumentation a limited number of functions to get cycle information, and insert probes before and after call to library functions. An important feature of this assembly probes mechanism is its ability to draw a much more 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 [4].

[3] Buffers should be picked large enough to limit calls to the C function preventing overhead and I/D-Cache pollution.

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

5.4 Value profiling

Value profiling is a critical add-on from MAQAO over traditional tools. Monitored values can be any registers singled out as being of higher potential:

- *Function parameters*, distribution of parameters values for any given function is a clear indicator that code versioning is an optimization to consider;
- *Addresses used in load/store instructions*, this allows to detect alignment problem, like bank conflicts or less known load/store queue conflicts [1]
- *ITC register*, direct access to this clock register allows a fine grain execution time profiling;
- *LC register*, this register is used for every counted loops to store the number of iterations to perform. This is a powerful parameter to evaluate relevance of SWP.

Prefetch distance is also interesting to monitor, for the moment we apply a simple analysis scheme. But the compiler can switch two address streams for the same `lfetch` instruction. This smart optimization usually occurs in short SWP loops. In such a case we can only compute the prefetch distance for one of these streams. A register dependency analysis module will be needed by MAQAO to handle this particular case.

5.5 Hardware monitor supports

There is few added value of MAQAO for hardware monitor support, basically it is an interface to use *perfmon* [12]. However since *perfmon* is a complex tool in this particular case a GUI is a interesting feature. This interface is implemented with calls to empty functions which are inserted within the application *perfmon* using them as trigger to start and stop monitoring. This allows to observe only the code fragment of interest. Incorporating the analysis tree described by Levinthal [15] to help end-user to navigate among counter terminology would be a nice feature.

Overall, MAQAO performs instrumentation at the assembly level, therefore with low disturbance of the real behavior of the application, low-overhead being a nice side effect. Thanks to its smart buffer scheme, it offers finer granularity of observation than other tool. At last value profiling establishes a link between the execution and the processed data.

6 Case Studies

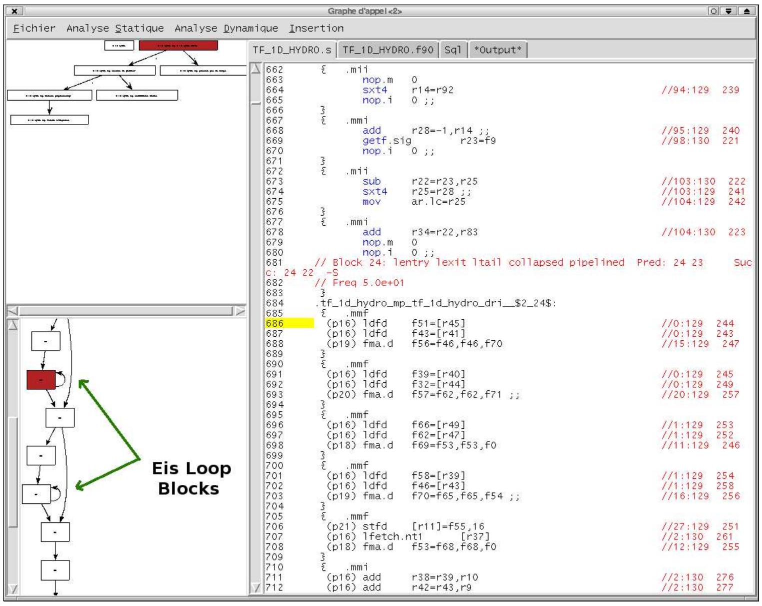
In this section we report results obtained using MAQAO to analyze and optimize two different applications. One being integer intensive with limited memory requirement while the second code is a more traditional scientific code focused on floating points performance and stressing the memory bandwidth.

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 L2 cache. On the software side we use Intel C/Fortran compiler 8.1 [5].

6.1 TERA Benchmark

The first case study concerns the optimization of the *TERA* [16] reference benchmark on the Itanium 2 processor. This benchmark is designed and used by the CEA-DAM (French atomic agency) and consists of the resolution of fluid dynamic equations with precise methods. Written in Fortran, two code sections deserve to be well optimized since they represent two of the most time consuming sections of the whole benchmark. They are of two kinds : the first one is a simple vector loop performing floating point intensive operations on several arrays whereas the second one involves a *while* structure controlled by array values and is composed of three vector loops. They are respectively called *Eis Loop* and *Totalisation.*

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



```
      {    .mii
              nop.m    0
              sxt4     r14=r92                                   //94:129   239
              nop.i    0 ;;
      }
      {    .mmi
              add      r28=-1,r14 ;;                             //95:129   240
              getf.sig         r23=f9                            //98:130   221
              nop.i    0 ;;
      }
      {    .mii
              sub      r22=r23,r25                               //103:130  222
              sxt4     r25=r28 ;;                                //103:129  241
              mov      ar.lc=r25                                 //104:129  242
      }
      {    .mmi
              add      r34=r22,r83                               //104:130  223
              nop.m    0
              nop.i    0 ;;
     // Block 24: lentry lexit ltail collapsed pipelined  Pred: 24 23     Succ: 24 22  -S
     // Freq 5.0e+01
      }
     .tf_1d_hydro_mp_tf_1d_hydro_dri__$2_24$:
      {    .mmf
       (p16) ldfd    f51=[r45]                                   //0:129   244
       (p16) ldfd    f43=[r41]                                   //0:129   243
       (p19) fma.d   f56=f46,f46,f70                             //15:129   247
      }
      {    .mmf
       (p16) ldfd    f39=[r40]                                   //0:129   245
       (p16) ldfd    f32=[r44]                                   //0:129   249
       (p20) fma.d   f57=f62,f62,f71 ;;                          //20:129   257
      }
      {    .mmf
       (p16) ldfd    f66=[r49]                                   //1:129   253
       (p16) ldfd    f62=[r47]                                   //1:129   252
       (p18) fma.d   f69=f53,f53,f0                              //11:129   246
      }
      {    .mmf
       (p16) ldfd    f58=[r39]                                   //1:129   254
       (p16) ldfd    f46=[r43]                                   //1:129   258
       (p19) fma.d   f70=f65,f65,f54 ;;                          //16:129   256
      }
      {    .mmf
       (p21) stfd    [r11]=f55,16                                //27:129   251
       (p16) lfetch.nt1      [r37]                               //2:130   261
       (p18) fma.d   f53=f68,f68,f0                              //12:129   255
      }
      {    .mmi
       (p16) add     r38=r39,r10                                 //2:130   276
       (p16) add     r42=r43,r9                                  //2:130   277
```

Figure 5: Control Flow Graph of the Eis Original Code Version. The upper left pane corresponds to the Procedure Call Graph of the file that contains the loop Eis. The bottom left pane corresponds to the Control Flow Graph of the function that contains Eis, and the two blocks targeted by the arrays corresponds to the assembly blocks implementing the software pipelined loop. The right pane displays the assembly source code of the red block selected in the bottom left pane.

6.1.1 Eis Loop

The first critical code section is the following one :

```
do i = first_cell, last_cell
  Eis(i) = Ets(i) - 0.5_r8*( Um(i)**2 + Um_p1(i)**2 + Um_p2(i)**2  )
end do
```

Two different implementations of this loop are compared : *Original* and *Improved*. *Original* consists of the original Fortran version compiled with the best compiling options (`-O3, -fno-alias`). *Improved* corresponds to an implementation trying to avoid the potential performance bottlenecks statically found by MAQAO.

Original Version MAQAO's static analysis allows to highlight three identified roots of bad performance just for this loop : deep software pipelined loop for the last iterations, costly array address computations and bad spill/fill register mechanism.

First, MAQAO's control flow graph (CFG, see Figure 5) of the loop reveals two different assembly implementations that may be executed sequentially, an illustration of loop fission. The first loop is 2-unrolled and SWP whereas the second one is only software pipelined. The loop analysis feature determines that the second loop is executed only when the vector sizes are odd. Even in this case, the loop performs only one iteration. This can lead to bad performance due to the high cost of the pipeline's prolog and epilog.

Secondly, the CFG also unveils the execution of a costly basic block for just performing array address computations, between the two loop implementations. On the Itanium 2, in Fortran, integer address computations are performed using high latency floating point instructions (such as `getf`, `setf`, `xma`). That severely increases the execution time for the last iteration (see Section 3.3). As a consequence, for odd vector sizes, the time spent only for the last iteration is equivalent to the execution of 14 iterations of the first loop implementation. This is a critical issue since MAQAO's instrumentation points out that this loop always computes 110 iterations for the benchmark reference test case.

Thirdly, explicit spill and fill mechanisms are found to be unnecessary since in the whole function not all registers are used (39 registers out of 128 remain at the end of the loop).

Improved Version Previous statical performance issues allow to generate an improved version of the loop.

First, porting the code from Fortran to C removes all address array computations performed with floating point units. Every computation is now performed using simple integer arithmetical instructions. Secondly, to avoid the cost of the last iteration, two versions are written in C: an 8-unrolled code and a loop tail code. For the 8-unrolled version, the compiler generates a pretty good SWP loop with a dense code (only 6% of `nop` operations) and makes an efficient usage of the large register file. The loop tail code, displayed in Figure 6, is implemented in C as a `switch` section in which each `case` is not followed by a `break`. This allows a late entry into the *switch-case* structure.

```
switch (nr) {
    case 7:
        Eis[n+6]=Ets[n+6]-0.5*(Um[n+6]*Um[n+6]+Um_p1[n+6]*Um_p1[n+6]+Um_p2[n+6]*Um_p2[n+6]);
    case 6:
        Eis[n+5]=Ets[n+5]-0.5*(Um[n+5]*Um[n+5]+Um_p1[n+5]*Um_p1[n+5]+Um_p2[n+5]*Um_p2[n+5]);
    // ... and so on for case 5 to 2
    case 1:
        Eis[n  ]=Ets[n  ]-0.5*(Um[n  ]*Um[n  ]+Um_p1[n  ]*Um_p1[n  ]+Um_p2[n  ]*Um_p2[n  ]);
}
```

Figure 6: Loop tail code for the improved version of Eis Loop

For this structure, the compiler generates a fully predicated code (7 iterations). Depending on the remaining iterations number, predicates are set such that only the required iterations are executed and retired. In this particular case it is far more efficient than the SWP version.

Overall these code transformations improve the performance of this loop by 22 %.

6.1.2 Totalisation Function

The other critical function of the benchmark is composed of a `while` structure with early exits, and three vector loops.

Original Version The code generated by the compiler is rather complex due to the highly conditional branching structure of this code section : the `while` structure contains exits only proceeded with `break` instructions. Moreover, inside this structure, the vector loops are easily located within the CFG. For each of them, MAQAO's analysis shows the compiler generates deep inefficient SWP loops for small vectors. They are preceded by a costly block of array address computations which in terms of execution time corresponds to 10 iterations. This is extremely high since MAQAO's instrumentation of this loop shows the loop trip count is constant for the whole execution of the benchmark and is set to 5 iterations. Although it is unknown by the compiler, the programmer knows this value since it corresponds to the number of variables used in the mathematical model ; this information can help the compiler generate a good code.

Improved Version Since this function is critical and heavily relies on branch structures and on the performance of the loops, a *loop peeling* is applied at high-level (C level) to improve the performance. It consists in generating dedicated codes for the most frequent iterations of the loop, in order to avoid entering in a costly software pipelined loop. Furthermore, pipeline depth is a crucial factor that determines the minimal number of iterations required to reach a steady state behavior and for these loops, it is inefficient for low trip counts. Such peeling transformation applied to one of the loop is represented in Figure 7.

First, the function fully rewritten in C removes all unnecessary and costly array address computations. Secondly, peeling is applied to all vector loops and on the `while` structure condition test. The `nb_var=5`

```
Original (Fortran):
  do j = 1, nb_var
    Mass( mn, j) = dxd*( Ro_p( ma, j) + 0.5*dxg*Dro_p( ma, j) )
  end do

Improved (C):
  if ( nb_var == 5 ){
    (*Mass1) = dxd*( Rop1[ma] + dxg*0.5*Drop1[ma]);
    (*Mass2) = dxd*( Rop2[ma] + dxg*0.5*Drop2[ma]);
    (*Mass3) = dxd*( Rop3[ma] + dxg*0.5*Drop3[ma]);
    (*Mass4) = dxd*( Rop4[ma] + dxg*0.5*Drop4[ma]);
    (*Mass5) = dxd*( Rop5[ma] + dxg*0.5*Drop5[ma]);}
  else {
    for ( j = 0 ; j < nb_var ; j++)
      Mass[mn+j*sMass1] = dxd*( Ro_p[ma+j*sRop1]+ 0.5*dxg*Dro_p[ma+j*sDrop1]);
```

Figure 7: Vector loop of Totalisation Function

case is then unrolled at high-level, which is converted into a predicated code giving better performance than the SWP version. For other `nb_var` values, a SWP loop is executed. Loop peeling technique adds several arcs in the control flow graph to select the variant to execute but improves the performance of the `nb_var=5` case. The *Improved* version gives better performance than the *Original* by about 20 %.

6.2 SHA-0 Attack Application

The second case study concerns the optimization of a cryptographic application: the SHA-0 attack on the Itanium 2 processor. This algorithm was developed by Chabaud and Joux [17] and this implementation was the first to find a full collision on SHA-0 in August 2004 [18]. Using MAQAO, we discover two main problems: the first one concerns the register allocation while the second one is related to memory/cache interaction.
To generate studied code, we found that the best compiling options are `-O2 -fno_alias`.

6.2.1 Program Overview

This program determines a full collision for SHA-0 algorithm with an algorithmic complexity of 2^{51}. In other words, this code finds 2 different messages having the same hashed value computed by SHA-0 algorithm and needs to test about 2^{51} messages before finding two colliding.
Written in C, the attack code is focused on integer computation. Indeed the algorithm manipulates integer value on 32 bits (because SHA-0 was designed for 32 bits architectures). The second characteristic of this code is complex control. For each pair of messages, the algorithm applies SHA-0 encryption turn by 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.

The attack code is separated in several parts but we are interested in the most time consuming one. This part (function called `do_neutral`) takes about 70 % of the whole CPU time and is in charge of comparing 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 computation made for previous messages.

6.2.2 Register file pressure

The first problem pointed out by MAQAO concerns the pressure on the register file. Event if, on Itanium 2, the number of integer registers available is quite large (128), the compiler seems to evaluate that more than 128 values are alive at a point. The first section provides results of MAQAO's static analysis (see Section 3.3) for the register allocation and the second one confirms the correctness of these static remarks.

Static Analysis MAQAO's static analysis highlights a potential bottleneck due to the high pressure on integer registers. Indeed the first remark is that the register file is fully used (saturated), according to the `alloc` instruction:

```
alloc r40=ar.pfs,2,87,7,8             // 32 registers always allocated + 96 requested
```

This not only means that the code needs all 128 integer registers but maybe even more. The compiler generates many implicit *spill/fill* instructions pointing out that, at some points, more than 128 registers are alive but looking at the assembly some of these instructions are superfluous.

```
{   .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 exact 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 8: Simple example of implicit *spill/fill*

In the code displayed in Figure 8, 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. But this is just an example on a short part of assembly instructions (only a few consecutive bundles). The same pattern appears across basic blocks putting the stress on the the potential problem in data flow estimation by the compiler. For example, the following *fill* occurs in a basic block:

```
add r31=344,sp              // Shift from the stack pointer
...
ld8 r44=[r31]               // Fill from the address sp+344
```

After this set of instructions, the next use of the register `r44` is about 600 lines after:

```
add r31=344,sp              // Shift from the stack pointer
...
st8 [r31]=r44               // Spill at address sp+344
```

Between these two points many traces are possible and not statically predictable (due to the complex control). Along each possible path between the *fill* and the *spill*, the value stored in this register is not used. Actually, in one path, `r44` is written but the function ends (with a return branch) just after. In that more complex case, the *fill* and *spill* instructions corresponding to `r44` can be removed.

Dynamic Analysis Replacing theses instructions with `nop`s or deleting (with rescheduling) them show that they were useless. Using memory fence and splitting `do_neutral` in smaller sub-functions are on the way to facilitate the register allocation for the compiler.

6.2.3 Memory/Cache Interference

As noted in program overview (see section 6.2.1), this code is focused on integer computation and contains mainly operations based on 32 bits integers (most of them bitwise). The second problem highlighted by MAQAO is related to memory interactions (loads/stores). In the following, we will see the results of static analysis concerning loads/stores instructions and the dynamic overhead of such interactions. According to these analysis we will see how to optimize the code in order to reduce memory/cache interferences.

Counter	Value
Cycles CPU	24 309 980
Numbers of instructions	79 620 526
Stalls Cycles	3 266 458
Stalls Cycles due to L1D access	1 173 888
L1D Misses	76 099
Number of loads	10 081 502
Number of stores	4 177 894

Table 1: Hardware counters on `do_neutral` function

Static Analysis Because SHA-0 algorithm uses exclusively 32 bits integers, the SHA-0 attack intensely uses 32 bits integers. Indeed, in our studied function (`do_neutral`), this program applies partial SHA-0 encryption to two messages at the same time comparing them during the encryption. On the generated code this is showed by many memory interactions on 32 bits. On Itanium 2 architecture, 32 bits interactions could cause a performance loss because of the potential 8B unalignment. Indeed in a 32 bits arrays, one cell out of two is not aligned on a 64 bits boundary.
Furthermore, MAQAO's static analysis detects the potential sources of memory interactions burst. The code Figure 9 extracted from the generated assembly shows the high number of 32 bits loads in consecutive bundles:

```
{   .mmi
      ld4 r8=[r70]
      ld4 r3=[r46]
      add r79=r61,r69
}
{   .mmi
      add r2=r58,r69
      add r30=r50,r69
      add r29=r53,r69 ;;
}
{   .mmi
      ld4 r28=[r71]
      ld4 r27=[r47]
      xor r26=r8,r3
}
{   .mmi
      add r72=r57,r64
      add r45=r98,r69
      add r25=r51,r69 ;;
}
{   .mmi
      ld4 r24=[r83]
      ld4 r23=[r82]
      xor r22=r28,r27
}
{   .mii
      add r36=r62,r64
      add r35=r97,r69 ;;
      shl r21=r24,5
}
```

Figure 9: Consecutive `ld4`

Dynamic Analysis Table 1 displays list of hardware counters related to cache/memory for one call of the `do_neutral` function:

Hardware counters confirm the important number of memory interactions. It seems that stalls occur due to the excessive pressure on L1D. This is surprising, since the overall memory footprint is around 5 KB,

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

Table 2: Hardware counters on part of `do_neutral` function: original version and optimized with padding

fitting perfectly in L1D.

Optimizations MAQAO's static analysis warns on a potential problem with memory interactions by 32bits (`ld4`/`st4`). Dynamic analysis with hardware counters put the stress on cache problems (potential large memory footprint). Taking into account this two information, we tried to change the load/store alignment. For that we used padding to make the load/store of each cell aligned on a 64 bits boundary.
We applied this optimization on a part of `do_neutral` function and we obtain a speed-up of 2 with the same compiling options (see Table 2).

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 L1 access. Notice that the number of instructions retired increases with the padding because instead of accessing directly the arrays, we need to multiply by 2 the indice.

7 Optimization

If a correct diagnosis of the problem is useful, what really matters is to find a solution/workaround for the problem detected. In this section, we briefly describe how the MAQAO tool could be used to interface with different optimization strategies. This part of the tool is still in the design phase and only key principles are given.

A first general (and well known) principle is that the optimization should be mainly targeting loops and profil guided. Once loops have been sorted by their contribution to execution time, optimization should be done following that order.

As an output, MAQAO can automatically produce a detailed summary on a per loop basis of the major "mistakes" done by the compiler and also an estimation of the gap between the theoretical bounds and the observed performance. This second metric is essential because in some cases, the compiler might have done mistakes but the overall impact (and therefore the maximal performance gain) could be shown to be less than 5%. Then this summary can be used to try to correct these mistakes.

First, let us describe a clever manner of correcting some. For example, for a given loop if some prefetch instructions are missing, a first solution would consist in inserting prefetch directives at the source level. A more appropriate solution is to use specific compiler options which would force the compiler to insert prefetch. A danger of such a solution when applied in a simple manner is that compiler options in general cannot target directly a specific loop. Therefore, one possible pitfall would be that some compiler options useful for some loops would be detrimental to other loops contained in the same file. A better solution would consist in extracting the "faulty" loop and wrap it into a subroutine on which the specific options could be used. This technique would work pretty well if simple mistakes are detected.

A sophisticated algorithm would rely on the detection of code templates and their automatic substitution by optimized versions [2]. Comparing two templates is not a trivial except when the control flow exhibits good properties (static, no aliasing, array indices as affine equations). A more brute force technique which could work in more complex cases would be again to isolate the "faulty" loop and export it in a separate file. Then on this file, a systematic search on the various compiler options could be performed: for each compiler option setting, code is produced and analyzed by MAQAO, which can detect the optimal compiler option settings.

Finally, MAQAO could be easily interfaced either with compilers like gcc or with tools such as Xemsys Library Generator [22] which are not full compiler but tools specialized in optimizing a specific class of loops such as simple vector loops.

8 Conclusion

As shown in the case studies MAQAO already fulfills needs as a convenient and powerful analysis platform. Addressing the performance problem at the assembly level seems relevant, especially on Itanium 2 due to the EPIC instruction set. The ability to process parsed code with a database allows end-user to build up incrementally his knowledge base. SQL offers a flexible interface to dive into the code structure tracking potential performance problem. A performance model extending Davidson's proposal model is also included. At last assembly level dynamic instrumentation performs value profiling as well as more common profiling tasks without altering the behavior of the application. It should be noted that all the analysis performed by MAQAO could be integrated within a compiler.

However, MAQAO is still in its development stages, and several limitations will be addressed. On the static side, additional analysis, like precise monitoring of prefetch distance, will be available when the data dependency module will be fully implemented. Another task is to extend and refine the MACS model, so that it could guide subsequent profiling to better exploit performance gaps. Obviously the main effort will be done on the optimization aspect.

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] Alexandre Farcy, Olivier Temam, Roger Espasa, Toni Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. MICRO 1998: 59-68

[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 ,Francois 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 Intel® Itanium® 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] Eric L. Boyd, Geith A. Abandah, Hsien-Hsin Lee and Edward S. Davidson, Modeling Computation and Communication Performance of Parallel Scientific Applications: A Case Study of the IBM SP2, Supercomputing '95

[14] Eric L. Boyd, Waqar Azeem, Hsien-Hsin Lee, Tien-Pao Shih, Shih-Hao Hung and Edward S. Davidson, A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1 Proceedings of the 1994 International Conference on Parallel Processing (ICPP), St Charles, il. 1994.

[15] 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

[16] 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

[17] 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,

[18] A. Joux, P. Carribault, C. Lemuet, W. Jalby. Full iterative differential collision in SHA-0, Preprint

[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