Detecting High-Level Performance Properties based on MAQAO and Periscope

Lamia Djoudi[1], Michael Gerndt[2] and William Jalby[1]

[1] Universite de Versailles
Laboratoire PRiSM, 45 avenue des Etats-Unis, 78035 Versailles Cedex, France
[2]Technische Universität München
Fakultät für Informatik I10, Boltzmannstr. 3, 85748 Garching, Germany
Tel: +33(0)1 39 25 43 43 - Fax +33 (0)1 39 25 40 57
Contact: lamia.djoudi@prism.uvsq.fr

Abstract

Tuning programs for high performance architectures requires a cyclic approach based on performance analysis and subsequent application of program transformations. Not only the communication and synchronization overheads of parallel programs have to be analyzed and reduced, but also the compute performance of each processor core.

While today's compilers already apply powerful program transformation to generate efficient code, frequently, additional hand tuning can improve it. Thus, performance analysis tools have to be available supporting the programmer in the detection of inefficient code and its optimization. Current tools are quite limited since they only perform measurements based on the processor's hardware counters.

This paper describes a new approach for performance analysis tools which combines static and dynamic information. We formalize inefficient single core execution in form of Periscope's performance properties. Detection of these properties is based on MAQAO's static information resulting from analyzing the program's assembler code, and on dynamic measurements taken during Periscope's automated online performance analysis.

Keywords Performance analysis, program tuning, program transformations, tools

1. Introduction

Performance analysis tools for parallel programs focus on the parallel execution and neglect single node performance. The best support they provide is the ability to perform measurements on the hardware performance counters and to present the measured values without any explanation to the programmer.

On the other hand, many HPC application are well tuned with respect to parallel execution but use at most 10% [1] of the compute core's peak performance. This gap is very significant and will probably become even more important as the processor's internal structure gets more and more complex.

Compilers, the main component in code generation for those processors, already are highly sophisticated. They contain a large number of optimizations. They can apply complex optimizations to provide independent

[1] http://www.lrz-muenchen.de/services/compute/hlrb/betriebszustand/performance/hlrb2_1week.html

machine instructions to out-of-order processors, for example, or to schedule instructions such that memory access latencies are hidden. These optimizations, often come from the research world, go through various stages of refinement and applicability, before being included in real world compilers. Most of the optimization process relies on heuristics. Sometimes, turning on all of the optimization flags results in a code which is slower than just turning on 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 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 can be detected on the assembly code but not with the help of the hardware performance counters.

Based on this insight we combine static and dynamic analysis within an automatic performance analysis process. First MAQAO performs an automatic static analysis and then the information is transferred to Periscope which searches for performance bottlenecks taking into account MAQAO's results.

MAQAO[13] analyzes plain assembly code and detects inefficiencies in the code due to various reasons, such as, failing optimizing transformations in the compiler or obscure compiler decisions.

Periscope [4] is an automatic performance analysis tool, that searches for predefined performance properties which are based on measurements during program execution. Combining those tools, thus enables a more powerful analysis.

The rest of the paper is organized as follows. In the next two sections, we introduce MAQAO and Periscope. Section 4 outlines our approach taken. Section 5 presents some of the new properties that have been written for Periscope based on MAQAO's analysis. Section 6 presents an experiment with a scientific code. The last two sections present related works and some conclusions.

2. MAQAO

Gathering data and statistics is necessary for a performance tool, but it remains only a preliminary stage. The most important step is to build a comprehensive summary for end-users and extract manageable information. MAQAO [13] stands for Modular Assembly Quality Analyzer Optimizer. It computes the structure of the assembly code, analyzes the application based on expert knowledge. Code pattern detection can be combined

with hotspot detection or other dynamic analysis. As a result, it has a great potential in revealing the possible flaws of the code generation. The concept behind this tool is to centralize all low level performance information and build correlations. As a result, MAQAO produces more and better results than the sum of the existing individual methods. Additionally, being based after the compilation phase allows a precise diagnostic of compiler optimisation successes and/or failures.

MAQAOPROFILE[10] is a MAQAO module which allows us to give a precise weight to all executed loops, therefore underscoring hotspots. Correlating this information provides the relevant metrics: (i) the hotpath at run-time which passes through the whole program and where the application spends the most of its time. (ii) the monitoring trip count is very rewarding. By default most of the compiler optimizations target asymptotic performance. Knowing that a loop is subjected to a limited number of iterations allows us to choose the optimizations characterized by a cold-start cost.

MAQAOAdvisor [11] implements a set of rules to help end-user to detect and understand performance problems. It helps end-users to navigate through the code and isolate the particularly important or suspicious pieces of code. For these isolated pieces which are the hot inner loops, *MAQAOAdvisor*: (1) suggests the optimizations, improves the code quality and the performance, (2) provides as many guidances as possible to help the decision making process. It can generate a report (file), assessing code quality and providing various performance hints for all loops found in the code. This report is the input file of Periscope.

3. Periscope

Periscope is a scalable automatic performance analysis tool currently under development at Technische Universität München. It consists of a frontend and a hierarchy of communication and analysis agents. Each of the analysis agents, i.e., the nodes of the agent hierarchy, searches autonomously for inefficiencies in a subset of the application processes.

The application processes are linked with a monitoring system that provides the Monitoring Request Interface (MRI). The agents attach to the monitor via sockets. The MRI allows the agent to configure the measurements; to start, halt, and resume the execution; and to retrieve the performance data. The monitor currently only supports summary information.

The application and the agent network are started through the frontend process. It analyzes the set of processors available, determines the mapping of application and analysis agent processes, and then starts the application. The next step is the startup of the hierarchy of communication agents and of the analysis agents.

After startup, a command is propagated down to the analysis agents to start the search. The search is performed according to a search strategy selected when the frontend is started [5]. The strategy defines an initial set of hypotheses, i.e., properties that are to be checked in the first experiment, as well as the refinement from found properties to a new set of hypotheses. The agents start from the set of hypotheses, request the necessary information for proofing the hypotheses from the monitor, release the application for a single execution of a repetitive program phase, retrieve the information from the monitor after the processes were suspended again, and evaluate which hypotheses hold. If necessary, the found hypotheses might be refined and the next execution evaluation cycle is performed.

At the end of the local search, the detected performance properties are reported back via the agent hierarchy to the frontend. The communication agents combine similar properties found in their child agents and forward only the combined properties.

Periscope starts its analysis from the formal specification of performance properties in the APART Specification Language (ASL) [2]. The specification determines the condition, the confidence value and the severity of performance properties. The analysis applied by Periscope for detecting inefficient single-node or better single-core performance is an excellent example for the incremental search. Optimizing single-core performance is critical for many applications since they usually reach only less than 10% of the peak performance of today's microprocessors.

The main source of information about the code's efficiency with respect to the execution pipeline and the cache hierarchies of the performance counters available in all processors. The Itanium 2 used in the Altix 4700 at Leibniz Computing Center provides a powerful hardware facility for performance monitoring. It has 12 counters capable of counting a large set of events. A very important feature is its support for counting the stall cycles occurring in the pipeline for different categories of events. These stall cycle counters are organized in a hierarchy allowing to drill down on the causes of low performance.

The root of the stall cycle counter hierarchy is the `BACK_END_BUBBLE_ALL` counter which determines the number of lost processor cycles. Comparing this to the total number of cycles gives the severity of inefficiencies in processor usage.

On the next level the following stall cycle counters are provided:

- BE_FLUSH_BUBBLE_ALL: pipeline flushes e.g. due to branch misprediction
- BE_L1D_FPU_BUBBLE_ALL: e.g. slow L1D hits and TLB misses
- BE_EXE_BUBBLE_ALL: inter-register dependencies or load instructions

- BE_RSE_BUBBLE_ALL: Register Stack Engine

- BACK_END_BUBBLE_FE: Instruction fetches

The example bellow shows a performance property formalized in ASL.

```
PROPERTY BackEndStallCycles(Perf s){
CONDITION:s.BACK_END_BUBBLE_ALL > 1;
CONFIDENCE:1;
SEVERITY:s.BACK_END_BUBBLE_ALL/phase_perf.cycles; }
```

The condition determines whether the property holds, i.e., if any stall cycles occur. The confidence is 1 since it is not only a hint for the property but a proven property based on measured performance data. The severity returns the percentage of the entire execution time of a single execution of the phase lost due to stall cycles.

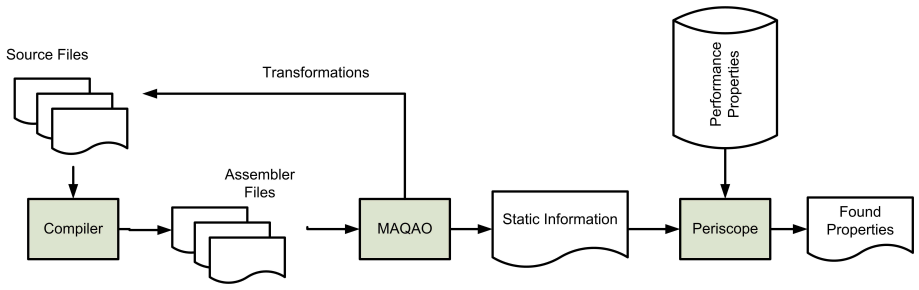

Figure 1. MAQAO and Periscope Integration

All the current properties in Periscope give the percentage of execution time lost by this property. This allows to rank all the found properties and let the programmer start optimizing the code for the worst one.

Properties are implemented in Periscope in form of C++-classes.

4. Approach

The approach we are taking here is to formalize performance properties that combine static information from analyzing the assembler code with MAQAO with dynamic information from the measurements during program execution by Periscope and to automatically search for these properties in Periscope.

The approach is shown in Figure 1. The target compiler generates the assembler files. These files are then analyzed by MAQAO which might already trigger some optimizations. MAQAO will then generate files with the results from the static analysis. These files are read by Periscope and the static information is taken into account in the automatic performance analysis.

The approach allows bridging the gap between the code analysis provided by MAQAO and the dynamic analysis performed automatically by Periscope.

5. Performance Properties based on MAQAO's information

In this section we give some examples for the new performance properties that are based on static information. The properties are structured into three groups: (1) Properties for individual instructions. (2) Properties for up to two bundles. (3) Properties for chuncks of bundles, e.g., entire innermost loops.

5.1 Properties for individual assembly instructions

This section includes properties that detect assembly code patterns based on single instructions. We associate with each property some hints concerning possible optimizations, depending on the compiler used. All the properties in this paper are in the context of the Itanium 2.

Conversion Integer Float: On itanium 2, the FPU is coupled to the integer data path via transfer paths between the integer and floating-point register files. Transferring values between floating-point and integer registers by means of the getf and setf instructions. These tend to show up particularly in the use of mod instructions where both arguments are variables. Also, address computations for multidimensional arrays are optimized via strength reduction techniques which avoids most of the integer multiplies. If this technique is not "correctly" applied, there might be some remaining integer multiplies to be performed. On Itanium 2, integer multiplies are very costly due to the lack of a specialized multiply integer unit: integers have first to be converted to floating point (using setf/getf instructions), then the the multiplication is performed by the FP multiply unit, and finally the result is converted back to integer.

```
PROPERTY ConversionIntegerFloat(Region r){
CONDITION: number(SETF/GETF) > 0;
CONFIDENCE:1;
SEVERITY: (((number(ANY)/number(SETF/GETF))* instances(ANY)
          *maxcycles(SETF,GETF))/cycles(phase)*100;   }
```

This property checks whether there are conversion instructions. `number(SETF)` returns the number of occurrences of SETF in the loop body and `number(ANY)` returns the number of instructions. `instances(ANY)` returns the number of issued instructions when executing the loop. `cycles(phase)` returns the number of cycles spent in the phase which is currently under inspection by Periscope. Thus, the severity first calculates the number of instances of SETF and GETF, multiplies this with the number of cycles for those operations and compares it to the entire execution time of the phase.

Inefficient loads and stores: Any instruction (load or store) accessing one, two or four bytes will have the same resources usage and cost as it's 8 bytes counter part. Therefore it always recommended, to try to pack low granularity accesses into eight bytes accesses.

```
PROPERTY Inefficient Loads){
CONDITION: instances(ld4 or ld2 or ld1) > 0;
CONFIDENCE:0.5;
SEVERITY:(instances(ld1)*cycles(ld1)*7/8 +instances(ld2)*cycles(ld2)*3/4 +
 instances(ld4)*cycles(ld4)/2)/cycles(phase);   }
```

The severity estimates the number of cycles that can be gained if the loads could be packed into 8 bytes loads. The confidences is only 0.5 since it cannot be guaranteed that the loads access subsequent addresses.

Advanced Speculative Load: Advanced loads and the corresponding correction instructions (check.s) are typically used when the compiler encounters a while loop structure or is unable to assert that two arrays refer to two non intersecting memory regions. The usual correction for the latter case is to force (if this is valid) the compiler to ignore dependencies between loads and stores, by using the -noalias flag for example.

```
PROPERTY Advanced Speculative Load){
 CONDITION: number(StoresALAT) >0
 CONFIDENCE: 1;
 SEVERITY:(instances(StoresALAT)*3)/cycles(phase)*100;   }
```

The severity is computed based on the instances of stores that are passed by the advanced load operations and therefore have to go to the ALAT table at runtime. The number of instances is calculated as in the `Conversion Integer Float` property. The additional check makes the stores more expensive, here estimated as 3 additional cycles for the ALAT check.

Missing Prefetch Property: Due to the latency of memory accesses it is very beneficial to prefetch data early enough so that they are available in the cache when they are accessed.

```
PROPERTY Missing Prefetch(Region r){
 CONDITION: number (ld) - number(PREFETCH) > 0
 CONFIDENCE: 0.5;
 SEVERITY: BE_EXE_BUBBLE_ALL/cylces(phase)*100;  }
```

This property checks whether there are more LD instructions in the given program region, e.g., an inner loop, then PREFETCH instructions. Whether the missing prefetches are really a problem depends on whether the

Accessed Banks	Performance in cycles
0 0 0 0 (quadruple conflicts on bank 0)	2
0 0 1 0 (triple conflicts on bank 0)	2
8 0 8 0 (two double conflicts on bank 0 and 8)	1.8
8 0 8 1 (double conflicts on bank 8)	1.8
8 0 9 1 (no conflict)	1

1. Bank Conflicts on the Itanium 2 Processor.

data are already in the cache. Thus the confidence is set to 0.5. Although not all of the stall cycles could be eliminated by prefetch operations, the severity is based on the number of stall cycles for load operations which can be measured with the counter `BE_EXE_BUBBLE_ALL`. This time is compared to the cycles of the entire phase of the execution for which Periscope checks the property.

5.2 Properties for two bundles of instructions

Potential L2 bank conflicts: It consists in counting couples of successive bundles with four memory instructions. On the Itanium 2 processor, 4 data accesses can be handled in one cycle if they target different banks in the L2 cache and, more precisely, provided there are no conflicts among them or among earlier issued operations. Nevertheless, these conflicts are local and may only occur between operations that are issued few cycles apart (between load and stores operations) or issued at the same cycle (in the case of load/load operations or store/store operations). The case will be very complex because many cases will be covered ranging from the ideal case for distinct conflict.

```
PROPERTY Potential bundle bank conflict){
 CONDITION: number(pairs of bundles with 4 loads)>0;
 CONFIDENCE:0.3;
 SEVERITY: instances(pairs of bundles with 4 loads)*2/cycles(phase)*100; }
```

The condition tests whether there are bundles in the loop body that might lead to a bank conflict. The severity is again based on the dynamic instances that are estimated as in previous properties. We calculate two additional cycles for a bank conflict.

5.3 Properties for entire loops

Bad Code Generated By Compiler: This property compares the number of issues with the theoretically required minimum number of issues. This theoretical minimum of issues is computed by MAQAO from the

LINE	SEVERITY	PROPERTY
657	19,0	Inefficient code generated by compiler
657	7,1	IA64 Pipeline Stall Cycles
657	5,8	Stalls due to waiting for data delivery to register
657	5,8	Missing prefetch instructions
657	4,7	Stalls due to waiting for FP register
657	4,1	L3 misses dominate data access
749	19,9	IA64 Pipeline Stall Cycles
749	18,4	Stalls due to waiting for FP register
749	15,8	L3 misses dominate data access
749	14,1	Stalls due to waiting for data delivery to register
749	14,1	Missing prefetch instructions
749	5,9	Stalls due to floating point exceptions or L1D TLB misses
749	5,5	Inefficient code generated by compiler
749	4,9	Stalls due to L1D TLB misses
749	4,8	Stalls due to hardware page walker
749	3,0	Advance speculative load instructions
780	11,9	Stalls due to waiting for FP register
780	11,1	IA64 Pipeline Stall Cycles
780	9,8	L3 misses dominate data access
780	8,3	Stalls due to waiting for data delivery to register
780	8,3	Missing prefetch instructions
780	4,1	Inefficient code generated by compiler

2. Single-node performance properties for subroutine VELO

MAQAO INFO	LINE			EXPLANATION
	657	749	780	
Issues	84	5	4	Number of issues estimated by compiler.
TheoBound	9	4	3	Minimal number of issues determined by MAQAO.
total_instr	504	30	24	Number of instructions in loop body.
IntFloat	0	0	0	Number of SETFGETF operations.
tload-rpref	23	8	6	Estimated number of non prefetched loads.
load.a	0	1	0	Number of stores going to ALAT.
4load	5	1	1	Number of pairs of bundles that might lead to bank conflict.

3. Information for the three loops given from MAQAO to Periscope.

number of memory, floating point, and integer instructions. It ignores dependencies and other reasons that can lead to NOP instructions in the bundles.

```
PROPERTY BadCodeGeneratedByCompiler(Region r){
 LET R1=number(issues) / theoretical_bound(issues)
 CONDITION: R1 > 1.2;
 CONFIDENCE: 0.1;
 SEVERITY: ((1-1/R1)*cycles(r))/cycles(phase)*100; }
```

The confidence is in general quite small since many important aspects are ignored. In software pipelined loops, due to the wealth of registers, the compiler is, in general, able to hide latencies between dependant instructions by scheduling other instructions in between. Therefore, the theoretical bounds (taking into account resource and issue unit usage) are fairly accurate and could be achieved by an appropriate instruction scheduling.

SWPLoop Dominated By Prolog Epilog: This property detects software pipelined loops where the cycles for the prolog and epilog dominate the execution. This can happens if the overall number of loop iterations is small. Code without software pipelining might be better. This property is an example for the third class related to software pipelining.

```
PROPERTY SWPLoopDominatedByPrologEpilog(Region r){
 CONDITION:  NumberOfIterations(r)<ar.ec-1;
 CONFIDENCE: 1;
 SEVERITY: OverheadSWPLoop/cycles(phase)*100; }
```

The register ar.ec in the Itanium 2 assembler code specifies the number iterations in the prolog and epilog of a software pipelined loop. If this is larger than the overall number of iterations, the loop is considered to be suboptimal.

6. Experiments

As an example, we present a performance analysis of CX3D, a crystal growth simulation code from Forschungszentrum Jülich [9]. It simulates the melting and crystallization process in the production of silicon wavers. The crucible is modeled as a three dimensional array, i.e, the radius, the height and the angles are individually discretized. For our experiments we used a discretization of (32, 92, 42) for (radius, angle, height). The radius of the crucible is 3 cm and its height 4cm. The application consists of a time loop in which the temperature distribution and the flow of the melt is simulated. The most time consuming part of the application taking more than 80% of the execution time is the subroutine `VELO`. It computes the new 3D velocity vectors in each grid point by iteratively solving the partial differential equation system.

Table 2 summarizes the most important properties found for the loops in VELO. The list contains properties that are purely based on Itanium's performance counters, such as `IA64 Pipeline Stall Cycles`, as well as those based on MAQAO's static information possibly combined with dynamic information, such as `Missing prefetch instructions`.

Table 3 presents the information provided by MAQAO to Periscope. This information is used in the properties presented in Section 5. The following code presents the loops for which the properties were detected by Periscope. The first one in source line 657 is a long loop that contains many operations and references. The other loops are shorter which is also reflected in the number of instructions shown in MAQAO's static information.

```
657 ----------------------------
       DO 4 K=2,N-1
        DO 4 J=2,M
         DO 4 I=2,LM1
             UN(I,J,K) = U(I,J,K) + DT * ((V(I,J,K) + V(I+1,J,K) + V(I,J
     X          -1,K) + V(I+1,J-1,K)) ** 2 * 0.0625D0 * RPI(I) - (R3(I+1
             ...
             WN(I,J,K) = W(I,J,K) + DT * (BT * (0.5D0 * (TT(I,J,K+1) +
       4 CONTINUE
749 ------------------------------------
        DO 105 K=2,N - 1
          DO 105 J=2,M
            do 105 i=2,LM1
            DCHECK = (R1(I) * UN(I,J,K) - R2(I) * UN(I - 1,J,K)) * U1I(
     *      I) + (VN(I,J,K) - VN(I,J - 1,K)) * V1I(I) + (WN(I,J,K) - WN(
            DP(I,J,K) = (-(BI(I) * DCHECK))
            P(I,J,K) = P(I,J,K) + DP(I,J,K)
       105  CONTINUE
780 ------------------------------------
      DO 106 I=2,LM1
        DO 106 K=2,N - 1
          DO 106 J=2,M
            UN(I,J,K) = UN(I,J,K) + DTDR * (DP(I,J,K) - DP(I + 1,J,K))
            VN(I,J,K) = VN(I,J,K)+RFI(I)*DTDPHI*(DP(I,J,K) - DP(I,J + 1,K))
      106 CONTINUE
```

7. Related Work

Assessing precisely quality of compiled code is essential to deliver high performance. Nowadays there is a lot of performance analysis tools/toolkits that focused on code analysis and optimization. But there are a very few tools focussing at providing user with transformation code advices for performance tuning. Tools such as foresys [14] or FORGExplorer [15] propose code analyses as well as code transformations but no techniques to identify the tuning transformation to use.

Davidson et al.[19], [20] 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. 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 useful for quantifying the quality of the code produced.

Several projects in the performance tools community are concerned with the automation of the performance analysis process. Paradyn's [6] Performance Consultant automatically searches for performance bottlenecks in a running application by using a dynamic instrumentation approach. Based on hypotheses about potential performance problems, measurement probes are inserted into the running program. Recently MRNet [7] has been developed for the efficient collection of distributed performance data. The Expert [8] tool developed at Forschungszentrum Jlich performs an automated post-mortem search for patterns of inefficient program execution in event traces. Potential problems with this approach are large data sets and long analysis times for long-running applications that hinder the application of this approach on larger parallel machines. Aksum [12], developed at the University of Vienna, is based on a source code instrumentation to capture profile-based performance data which is stored in a relational database. The data is then analyzed by a tool implemented in Java that performs an automatic search for performance problems based on JavaPSL, a Java version of ASL.

8. Conclusions

Performance analysis and optimization is an important part in the code development for high-performance systems. The environment with a combination of MAQAO and Periscope allows to automatically search for predefined performance properties based on static and dynamic information.

The combined performance analysis environment is currently under development by the University of Versailles and Technische Univeristät München. The current target architecture is the Itanium 2 processor which is used in the Altix 4700 supercomputer at Leibniz Computing Centre.

While we currently concentrate on relative local code regions, i.e., inner loops, we plan to extend the work to outermost loops and entire functions. In addition, we plan to feedback the dynamic information gathered by Periscope into MAQAO to support its intelligent optimization module.

References

[1] Vtune performance analyzers. *www.intel.com/software/products/vtune/*.

[2] T. Fahringer, M. Gerndt, G. Riley, and J. Träff. Knowledge specification for automatic performance analysis. *Technical Report, www.fz-juelich.de/apart*, 2001.

[3] T. Fahringer and C. Seragiotto. Aksum: A performance analysis tool for parallel and distributed applications. *Performance Analysis and Grid Computing, Eds. V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller, Kluwer Academic Publisher, ISBN 1-4020-7693-2, pp. 189-210*, 2003.

[4] M. Gerndt and K. Fürlinger. Specification and detection of performance problems with ASL. *Concurrency and Computation: Practice & Experience*, 19(11):1451 – 1464, August 2007.

[5] Michael Gerndt and Edmond Kereku. Search strategies for automatic performance analysis tools. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007*, volume 4641 of *LNCS*, pages 129–138. Springer, 2007.

[6] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer, Vol. 28, No. 11, pp. 37-46*, 1995.

[7] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 Conference on Supercomputing (SC 2003)*, Phoenix, Arizona, USA, Nov 2003.

[8] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 13 - 22*, 2003.

[9] M. Mihelcic, H. Wenzl, H. Wingerath. Flow in Czochralski crystal growth melts. *Technical Report Jül-2697*, Forschungszentrum Jülich, 1992.

[10] L. Djoudi, D. Barthou, O. Tomaz, A. Charif-Rubial, JT. Acquaviva and W Jalby The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module *Workshop on EPIC*, San Jose, 2007.

[11] L. Djoudi, J. Noudohouenou and W. Jalby The Design and Architecture of MAQAOAdvisor: A Live Tuning Guide *international conference on high performance computing (HiPC) 2008, India*

[12] T. Fahringer and C. Seragiotto. Aksum: A performance analysis tool for parallel and distributed applications. *Performance Analysis and Grid Computing, Eds. V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller, Kluwer Academic Publisher, ISBN 1-4020-7693-2, pp. 189-210*, 2003.

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

[14] FORESYS, FORtran Engineering SYStem. http://www.pallas.de/pages/foresys.htm.

[15] FORGExplorer. http://www.apri.com/

[16] VTune Performance Analyzer http://www.intel.com/software/products/vtune

[17] A. Monsifrot and F. Bodin Computer aided hand tuning (CAHT): applying case-based reasoning to performance tuning *ICS '01: Proceedings of the 15th international conference on Supercomputing, 2001*

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

[19] 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

[20] 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.