John von Neumann Institute for Computing


# Periscope: Advanced Techniques for Performance Analysis

M. Gerndt, K. Fürlinger, E. Kereku

published in

*Parallel Computing:*
*Current & Future Issues of High-End Computing*,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 15-26, 2006.



http://www.fz-juelich.de/nic-series/volume33

# Periscope: Advanced Techniques for Performance Analysis*

Michael Gerndt [a], Karl Fürlinger[a], Edmond Kereku[a]

[a]Technische Universität München, Fakultät für Informatik 10, Email: gerndt@in.tum.de

Performance analysis of applications on supercomputers require scalable tools. The Periscope environment applies a distributed automatic online analysis and thus scales to thousands of processors. This article gives an overview of the Periscope system, from the performance property specification, via the search process, to the integration with two monitoring systems. We also present first experimental results.

## 1. Introduction

Supercomputers with up to hundreds of teraflops are currently build and deployed to solve grand challenge applications. These systems consist of thousands of processors arranged in a clustered SMP architecture, i.e., the basic building block is an SMP system with a small number of processors. Applications executed on such systems should be as efficient as possible due to the enormous costs of the machines and due to the goal of enabling new research results through high performance computing.

A major step in developing applications for such machines is therefore performance analysis and tuning. Optimizing an application for best performance already in the design phase is almost impossible due to the complex architecture of such machines. Actual program runs with real data and the desired number of processors have to be analyzed to detect performance problems and their causes. Scaling down the application while keeping the performance characteristics unchanged is almost impossible.

Our Periscope performance analysis environment supports the programmer in analyzing the performance of full size application runs. The major challenge solved by the tool is scalability to a large number of processors. This is possible by performing an automatic distributed online analysis. Analysis agents are distributed across all SMP nodes assigned to the application. Each agent automatically searches for performance problems of the processes and threads running on the SMP node. Low-level performance data are analyzed locally and detected performance problems are communicated through the network to the master agent which interacts with the programmer.

Periscope is based on a set of techniques developed in the APART working group on automatic performance analysis tools. The potential performance problems are formalized in ASL (APART Specification Language) [ 2] [ 3] and the interface between the monitoring system and the node agents is an extension of MRI (Monitoring Request Interface) [ 7]. The agents take into account also the program's structure, e.g., the nesting of program regions. This information is provided in SIR (Standard Intermediate Program Representation) [ 14].

This article gives an overview of Periscope and presents first results from real experiments. Section 2 outlines related work and Section 3 the APART Specification Language. The overall design of the Periscope system is presented in Section 4. It covers also the integration of different monitoring systems into Periscope. In the center of Periscope are the search strategies which are described in Section 5. Section 6 gives a usage scenario and Section 7 presents actual results from applying Periscope to the NAS Parallel Benchmarks and an application code. The article closes with a short

*This work is funded by the German Science Foundation (DFG).

```
PROPERTY LC1ReadMissesOverMemRef(SeqPerf s){
CONDITION:
  s.lc1_data_read_miss/s.read_access > 0.01;
CONFIDENCE:
  1;
SEVERITY:
  s.lc1_data_read_miss/s.read_access; }
```

Figure 1. This performance property identifies significant L1-cache miss rate.

summary and outlook in Section 8.

## 2. Related Work

Several projects in the performance tools community are concerned with the automation of the performance analysis process. Paradyn's [ 11] 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 [ 13] has been developed for the efficient collection of distributed performance data. However, the search process for performance data is still centralized.

The Expert [ 15] tool developed at Forschungszentrum Jülich 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 [ 1], 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.

## 3. APART Specification Language

Periscope starts its analysis from the formal specification of performance properties in the APART Specification Language (ASL) [ 4]. The specification determines the condition, the confidence value and the severity of performance properties. Beyond the individual specification of each property, the ASL provides property templates for specifying similar performance properties in a compact way and *metaproperties* for combining already defined properties. These techniques lead to extremely compact specifications.

The example in Figure 1 demonstrates the specification of performance properties with ASL. The performance property shown here identifies a region with high L1-cache miss rate. The data model, specified in ASL too, contains a class `SeqPerf` which contains a reference to a program region, a reference to a process, and a number of cache-related and other metrics. Figure 2 shows the C++-classes generated from the ASL specification. The instance of `SeqPerf` available in the property is called the property's context. It defines the program region, e.g., a function, and the process for which the property is tested.

The condition determines whether the property holds. It accesses information in the data model.

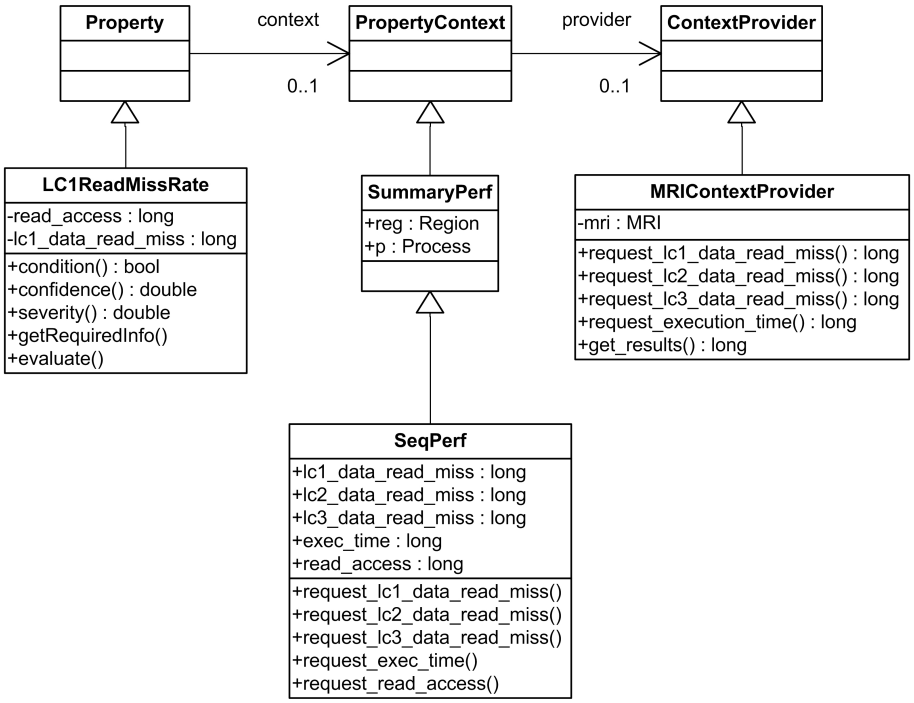


Figure 2. This figure presents the C++-classes generated from the ASL specification in Figure 1.

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 miss rate and allows to rank similar properties for all program regions and processes. Condition, confidence, and severity are implemented by methods of the C++-class generated from the ASL specification.

## 4. Periscope Design

Periscope consists of a graphical user interface based on Eclipse, a hierarchy of analysis agents and two separate monitoring systems (Figure 3).

The graphical user interface allows the user to start up the analysis process and to inspect the results. The agent hierarchy performs the actual analysis. The node agents autonomously search for performance problems which have been specified with ASL. Typically, a node agent is started on each SMP node of the target machine. This node agent is responsible for the processes and threads on that node. Detected performance problems are reported to the master agent that communicates with the performance cockpit.

The node agents access a performance monitoring system for obtaining the performance data required for the analysis. Periscope currently supports two different monitors, the *Peridot monitor* [ 5] developed in the Peridot project focusing on OpenMP and MPI performance data, and the *EP-Cache monitor* [ 10] developed in the EP-Cache project focusing on memory hierarchy information. Both monitors are described in more detailed in the following subsections.

The node agents perform a sequence of experiments. Each experiment lasts for a program phase, which is defined by the programmer, or for a predefined amount of execution time. Before a new experiment starts, an agent determines a new set of hypothetical performance problems based on the

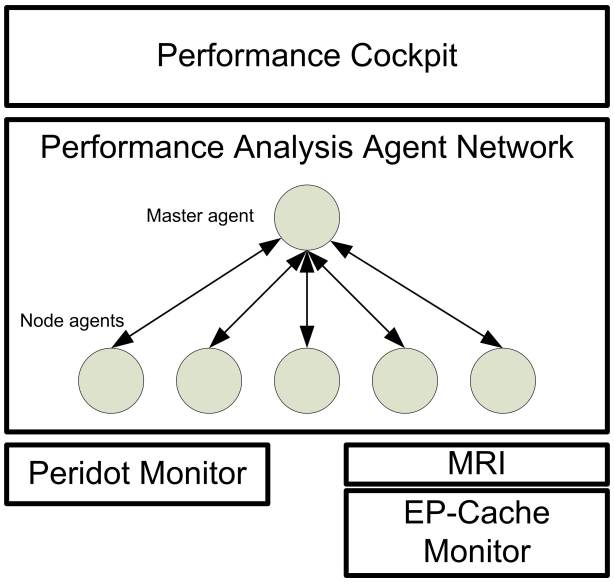


Figure 3. Periscope currently consists of a GUI based on Eclipse, a hierarchy of analysis agents, and two separate monitoring systems.

predefined ASL properties and the already found problems. It then requests the necessary performance data for proving the hypotheses and starts the experiment. After the experiment, the hypotheses are evaluated based on the performance data obtained from the monitor. Subsection 5 gives more details on the currently implemented search strategies.

### 4.1. Periscope Monitors

Although Periscope can be adapted for other monitoring systems, we currently use two monitors that where developed in two previous projects. The Peridot monitor includes new techniques to reduce the measurement overhead by offloading some of the tasks in monitoring to an additional monitoring process. The EP-Cache monitor reduces overhead by supporting detailed monitor configuration with the goal to measure only what is really necessary.

#### 4.1.1. Distributed Monitoring

The first monitoring system was developed in the Peridot project funded by KONWIHR. The target system was the Hitachi SR8000 installed at LRZ München. Its building blocks are 9-way SMP systems where 8 processors are available to user programs and the ninth is used by the OS.

The Peridot Monitor is shown in Figure 4(a). The node agent accesses the *Runtime Information Producer (RIP)* which is executed on a separate processor, i.e., on the Hitachi it is the OS processor of a node. In this figure processor 1 is one of the application processors. The application is linked with a monitoring library which has very little overhead since data are efficiently placed in an circular event buffer. No control decisions or aggregation operations are executed by the monitoring library. It is the task of the RIP to perform any time consuming operations.

The event buffer is placed in shared memory of an SMP node so that all application processes can deliver their performance data and the RIP can also access the buffer. In addition to utilizing physical shared memory, the implementation also provides support for the Remote Memory Access

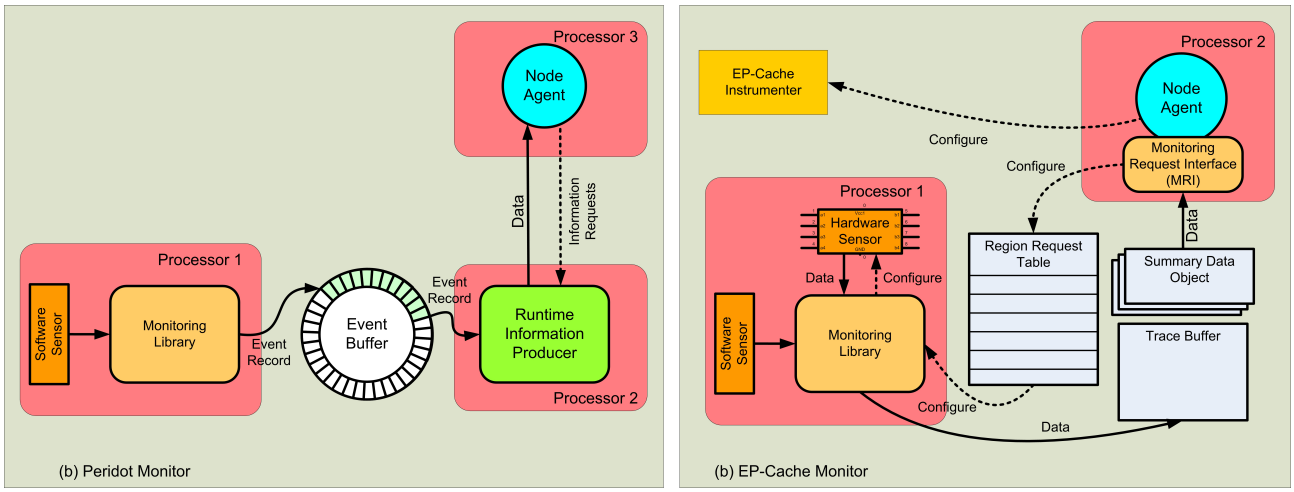


Figure 4. (a) Distributed Monitoring with the Peridot Monitor (b) Configurable Monitoring with the EP-Cache Monitor.

hardware of the Hitachi. Thus, the RIP can be executed on another SMP node and still efficiently access the event buffer.

The node agent accesses information in a pull model, thus it sends a request and receives the available performance data. Calls to the Peridot Monitoring Library are inserted by source code instrumentation with OPARI [ 12], via the MPI profiling interface, as well as by the function call instrumentation capability of GNU compilers (the `-finstrument-functions` compiler option).

### 4.2. Configurable Monitoring

The second monitoring system is the EP-Cache monitor in Figure 4(b). The goal of the EP-Cache project was to investigate new techniques for hardware monitoring of accesses to the memory hierarchy. The project developed a design of a hardware monitor that can monitor address ranges. This, for examples, allows to count only cache misses that occur for a specific array. In addition, the monitor can be configured to provide histogram information for address ranges with a predefined granularity, e.g., multiples of cache lines.

To be able to utilize this monitor's features effectively, the monitoring library had to support online configuration of the hardware monitor. For example, different data structures should be monitored in different loops of the program. This requires to reconfigure the monitor for the loop to be executed. We extented the configuration support for the hardware monitor also for software sensors that are inserted into the soure program by a source-to-source instrumentation tool called *EP-Cache Instrumenter* [ 8]. Since the monitoring overhead also depends on the amount of instrumentation, in principle, Periscope could also guide the instrumentation process, as shown in Figure 4(b).

To enable configuration requests for program regions, the instrumenter generates a file with information about the program structure. It utilizes the *Standard Intermediate Program Representation* (SIR) [ 14] developed in the APART working group. It is an XML notation for specifying - besides other information - program regions, their nesting structure, as well as the data structures accessed in the regions. This SIR file is read by Periscope so that it knows about the program regions and data structures and can use this information in the performance property search.

The node agent communicates with the EP-Cache Monitor via the *Monitoring Request Interface* (MRI) [ 7]. The MRI is based on a proposal in the APART group but was extended for the special needs in EP-Cache. MRI defines a set of operations to send measurement requests to the monitor. For example, the node agent can request measurement of L1 cache misses for data structure `A` in a specific loop. MRI also provides operations to retrieve the measured information for a specific request. The last functionality in MRI is to control program execution. The program can, for example, be started for the next program phase (see Section 5) by specifying the end of a program region which will automatically stop the program. Thus, the node agent can specify requests, start the execution of the next phase, retrieve and evaluate the performance data, and take decisions for the execution of the next phase (an experiment).

The requests as well as the resulting performance data, either aggregated information or trace information, are exchanged between the node agent and the application processor via a shared memory area.

### 4.3. Adapting Periscope to other Monitors

Currently, Periscope can make use of the Peridot and the EP-Cache monitor. While the first one allows measurements for OpenMP and MPI, the second one provides support for detailed measurements of the memory access behavior of OpenMP programs.

The two monitoring systems demonstrate that Periscope can be adapted to different monitoring systems. The following tasks are necessary to base Periscope on yet another monitor:

1. Define the ASL data model

   The ASL data model used in the specification of the performance properties (see Section 3) has to include only performance data that can be measured with the underlying monitor. Thus, the data model and probably the properties have to be adapted to the new monitor.

2. Develop a context provider

   Figure 2 shows that a property context is connected to a context provider. The context provider implements the connection to the monitor. In the figure, the `MRIContextProvider` class implements the connection to an MRI-based monitor, e.g., the EP-Cache monitor. It provides methods to request measurements as well as to access the measurement results. The same operations would have to be implemented for a class designed for another monitor, but only for those data that are included in the data model.

## 5. Search Strategies

The analysis agent is based on a small set of databases.

1. **Application Structure**: It stores the structure of the application which is available to Periscope through the SIR file generated by the F90 instrumenter.

2. **ASL Property**: It contains the C++-classes generated from the ASL property specification. Properties can be added to the database via shared libraries without recompilation of Periscope.

3. **Performance Data**: It stores the performance data retrieved from the monitor. It is based on the classes generated from the ASL performance model.

4. **Performance Problems**: This database stores the detected performance problems.

5. **Search Strategies**: Periscope currently provides several search strategies. The strategies determine which performance property hypotheses are evaluated in the next experiment.

As mentioned above, the whole search for performance problems happens online in a distributed fashion. The overall goal is to enable performance analysis for large scale applications.

The analysis executes a series of *experiments*. These experiments can be individual runs of the application as well as executions of subsequent phases during a single program run. Each experiment consists of several steps:

1. Hypothesis selection

   Based on the set of already detected performance problems, the structure of the application, the characteristics of the phase (cyclic of execute once), and on the limits of the monitoring system the next set of hypothesis to be tested is selected.

2. Monitor configuration

   The monitor is configured to deliver the performance data required to evaluate the performance hypotheses.

3. Experiment execution

   The application is released until it stops and the control goes back to Periscope.

4. Retrieving results

   The measured performance data need to be fetched from the monitor. The data are inserted into the performance data base, e.g., in the `SeqPerf` object. This step is implemented by the `getResults` method of the context provider.

5. Evaluation of hypothesis

   The analysis agent calls the `condition` method of the properties to check their existence. This method accesses the data in the performance model that were inserted in the previous step.

Periscope currently supports a small set of strategies for determining the performance problems. These strategies are a non-phase-based strategy and two phase-based strategies.

The *non-phase-based strategy* periodically evaluates all possible hypotheses. This strategy makes only sense in combination with the Peridot monitor, since explicit configuration is not necessary. Thus the monitor configuration step is omitted. After a predefined amount of time, the node agent retrieves the performance data and evaluates the hypotheses. In the next experiment, the set of hypotheses is adapted and evaluated after the next time interval.

The two phase-based strategies depend on the phase concept. The application is split into phases, called *application phases*. More precisely, a phase is a specific program region. The overall execution of the program thus includes a sequence of instances of those phases. Different applications are constructed with different phases. One possible example is depicted in Figure 5. Once the user identified those phases, appropriate measurements can be performed during the execution of those phases. In many of the HPC applications there is a time loop which consumes most of the execution time. This loop's body is an excellent example for a repetitive application phase.

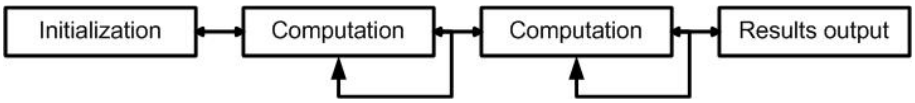


Figure 5. An example application including an initialization phase, several computational phases(which can be repetitive), and a concluding phase for possible deallocation and results gathering.

Periscope can basically use each instrumented code region as an application phase. While the phases are mainly intended to support online analysis, they can also be used to support iterative analysis via repetitive execution of the program. The phase in this case is the program's main routine.

The user may want to mark other code regions than standard code regions as application phases. Standard regions are, for example, functions, sequential loops, and parallel regions in OpenMP. We implemented a simple mechanism in our instrumenter, which allows specifying those phases everywhere in the application with very little effort. It only requires the user to insert "USER REGION" directives in the code before the instrumentation.

The first phase-based strategy gives higher priority to the identification of the location of performance problems. It first searches for basic performance properties, such as a significant L1- and L2-cache miss rate. It starts with the main program and the call sites in the main program. If a call site shows a high cache miss rate, it checks hypotheses for the outermost regions in that function. In the following experiments it checks for nested regions as well as for specific data structures that are responsible for the miss rate.

The incremental search for the location of a performance problem is required, e.g., due to the limited number of counters in the hardware monitor or due to the instrumentation overhead for fine grained program regions.

After the detection of the location, the search strategy refines in the performance properties. For example, the high miss rate can result from read or write misses, or from conflict and - in the case of multiprocessors - from coherence misses.

The second phase-based strategy gives priority to the refinement in the property hierarchy. Thus, before it refines in the regions and data structures, it will search for all possible performance problems for the regions currently under investigation.

## 6. Usage Scenario

To illustrate a usage of our analysis framework consider the following usage scenario, where a user wants to conduct an performance analysis of his MPI-based application based on the non-phase-based search strategy using the Peridot monitor. Note that this just represents one possible application scenario of our tool, a usage for OpenMP-based applications or using a batch system instead of interactive startup is possible too, for example.

1. Preparation of the application for performance analysis. In order to enable performance analysis of the application it has to be instrumented to enable the recording of performance data by our monitoring systems. Currently we support OpenMP monitoring based on the Opari [ 12] source-to-source instrumenter, MPI monitoring based on the MPI performance monitoring interface, and function-call monitoring based on the function call instrumentation capability

of GNU compilers (the `-finstrument-functions` compiler option). The instrumented application is then compiled and linked against our monitoring library.

2. Interactive startup of the application. The user starts his application using the `mpirun` command on a number of nodes of the system. The application begins to execute and on the first call into our monitoring library (typically, the `MPI_Init` call), the library performs the necessary initialization procedures like allocating buffers for the generated performance data. Furthermore the monitoring library registers itself with a registry service that is used by the components in our system to discover each other. At this point, the monitoring library halts the execution of the target application (unless a special environment variable is set) to allow the agent system to be started to monitor the execution of the application from the start.

3. The user interactively starts the performance analysis system by executing `./periscope <name of the application>`. The startup component then contacts the registry service to determine on which nodes of the system the application is executing and starts a node agent on each of the identified nodes.

4. After all node agents have been started a master agent is started on the interactive node and a `start` command is issued by the master agent to all node agents. This causes the halted application to be resumed.

5. Performance data is now generated by the target application and processed by the node agents according to the search strategy. Specifically, property contexts are instantiated as demanded for the evaluation of the properties to be checked. The `MPIOverhead` property, for example, is evaluated on `SeqPerf` contexts. Therefore, the node agents access the RIP (via the `RIPContextProvider`). The node agent can then apply the `MPIOverhead` property which in turn internally accesses the MPI data member of the `SeqPerf` class.

6. The discovered performance properties are communicated to the master agent that reports them to the user.

## 7. Results

As a first example, we present here the performance analysis of a crystal growth simulation code. It simulates the melting and crystallization process in the production of silicon wavers. The application consists of a time loop in which the temperature distribution and the flow of the melt is simulated. This is a sequential version of the code where Periscope searches for memory hierarchy problems.

In the first experiment, i.e., the first execution of the time loop, the call sites in the time loop are measured, i.e., calls to procedures CURR, VELO, TEMP, and BOUND. The results of the experiment are the following performance problems:

```
LC1MissesOverMemRef
  Severity: 0.156674
  Region: BOUND( CALL, main.f, 69 )
LC1MissesOverMemRef
  Severity: 0.0225174
  Region: TEMP( CALL, main.f, 68 )
```

The most severe performance problem is found in procedure BOUND which performs the bound-

aries update. In the next experiment the regions in BOUND are analyzed. This results in the following performance properties:

```
LC1MissesOverMemRef
  Severity: 0.68164
  Region: ( LOOP, bound.f, 139 )
LC1MissesOverMemRef
  Severity: 0.216865
  Region: ( LOOP, bound.f, 673 )
```

The next two experiments investigate the precise type of misses and the data structures responsible. The final list of performance problems found in that program run includes the following problems:

```
LC1WriteMissesOverMemRef (LOOP, bound.f, 139 )
  Severity: 0.671016
  Data Structure: UN
LC1MissesOverMemRef (LOOP, bound.f, 28 )
  Severity: 0.103128
  Data Structure: UN
LC1ReadMissesOverMemRef (LOOP, bound.f, 673 )
  Severity: 0.108437
LC1WriteMissesOverMemRef (LOOP, bound.f, 673 )
  Severity: 0.108429
```

The first property shows that the misses are almost all write misses due to array UN. The next property points to a problem with cache misses for data structure UN in loop 28 (This is the line number in file bound.f). But no more specific properties could be determined for that region. In contrast to the result for this loop, Periscope was able to prove that read and write misses have the same severity in loop 673 but no single data structure is responsible for the misses.

We also applied Peridot to the NAS Parallel Benchmarks, some application programs from the EP-Cache project, and the APART Test Suite [ 6]. Figure 6 shows the number of performance properties found for an OpenMP version of the NAS parallel benchmarks.

## 8. Summary and Future Work

This paper gives on overview about the Periscope system for automatic performance analysis of applications on teraflop computers. Scalability to execution runs on thousands of processors is achieved by executing an online distributed analysis. Analysis agents are autonomously searching for performance problems of a small subset of the threads or processes and reporting the problems back to a master agent.

In our experiments a single node agent is responsible for an SMP node of the target machine. Other configurations can certainly be implemented if the requirement of the communication interface with the monitoring system, i.e., communication via shared memory, is available.

This article presents also some first results applying Periscope to an application code and to the NAS Parallel Benchmarks. Periscope can search for a predefined set of performance properties. In these programs, either a database with properties for memory hierarchy problems is used or a database with performance problems for OpenMP programs.

| Property | BT | CG | EP | FT | IS | LU | MG | SP |
|---|---|---|---|---|---|---|---|---|
| ImbalanceAtBarrier | | | | | 1 | 3 | | |
| ImbalanceInParallel-Sections | | | | | | | | |
| ImbalanceInParallelLoop | 12 | 13 | 1 | 8 | 2 | 9 | 12 | 16 |
| ImbalanceInParallel-Region | 6 | 9 | 1 | | 2 | 8 | 2 | 5 |
| UnparallelizedInSingle-Region | | | | | | 3 | | |
| UnparallelizedInMaster-Region | 4 | | | | | 13 | 2 | 5 |
| ImbalanceDueToNotEnough-Sections | | | | | | | | |
| ImbalanceDueToUneven-SectionDistribution | | | | | | | | |
| CriticalSectionContention | | | | | | | | 1 |
| LockContention | | | | | | | | |

Figure 6. This table shows the number of performance problems found in the OpenMP version of the NAS parallel benchmarks.

We do not yet provide a nice GUI for Periscope, but we are implementing a GUI based on Eclipse. It will be used for phase specification, configuration of the search process, as well as for presenting and explaining the performance problems found.

Finally the environment can be used for performance analysis of Grid applications [ 9]. It is important in this context too, to do local analysis and to propagate only small amounts of high level information among the analysis agents.

## References


[1] T. Fahringer, C. Seragiotto: *Aksum: A Performance Analysis Tool for Parallel and Distributed Applications*, Performance Analysis and Grid Computing, Eds. V. Getov, M. Gerndt, A. Hoisi, A. Malony, B. Miller, Kluwer Academic Publisher, ISBN 1-4020-7693-2, pp. 189-210, 2003

[2] T. Fahringer, M. Gerndt, G. Riley, J.L. Träff: *Specification of Performance Problems in MPI-Programs with ASL*, International Conference on Parallel Processing (ICPP'00), pp. 51 - 58, 2000

[3] T. Fahringer, M. Gerndt, G. Riley, J.L. Träff: *Formalizing OpenMP Performance Properties with the APART Specification Language (ASL)*, International Workshop on OpenMP: Experiences and Implementation, Lecture Notes in Computer Science, Springer Verlag, Tokyo, Japan, pp. 428-439, 2000

[4] T. Fahringer, M. Gerndt, G. Riley, J. Träff: *Knowledge Specification for Automatic Performance Analysis*, APART Technical Report, www.fz-juelich.de/apart, 2001

[5] K. Fürlinger, M. Gerndt: *Peridot: Towards Automated Runtime Detection fo Performance Bottlenecks*, High Performance Computing in Science and Engineering, Garching 2004, pp. 193-202, Springer, 2005

[6] M. Gerndt, B. Mohr, J. Larsson-Traeff: *Evaluating OpenMP Performance Analysis Tools with the APART Test Suite*, Fifth European Workshop on OpenMP (EWOMP '03), RWTH Aachen, pp. 147-156, 2003

[7] M. Gerndt, E. Kereku: *Monitoring Request Interface Version 1.0*, TUM Technical Report, 2003

[8] M. Gerndt, E. Kereku: *Selective Instrumentation and Monitoring*, International Workshop on Compilers for Parallel Computers (CPC 04), 2004

[9] M. Gerndt: *Automatic Performance Analysis Tools for the Grid*, Concurrency and Computaton: Practice

and Experience, Vol. 17, pp. 99-115, 2005

[10] E. Kereku, M. Gerndt: *The EP-Cache Automatic Monitoring System*, International Conference on Parallel and Distributed Systems (PDCS 2005), 2005

[11] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995

[12] A. Malony, B. Mohr, S. Shende, F. Wolf: *Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting*, EWOMP 01, Third European Workshop on OpenMP, 2001

[13] Ph. C. Roth, D. C. Arnold, B. P. Miller: *MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools*, SC2003, Phoenix, November, 2003

[14] C. Seragiotto, H. Truong, T. Fahringer, B. Mohr, M. Gerndt, T. Li: *Standardized Intermediate Representation for Fortran, Java, C and C++ Programs*, APART Working Group Technical Report, Institute for Software Science, University of Vienna, Octorber, 2004

[15] F. Wolf, B. Mohr: *Automatic Performance Analysis of Hybrid MPI/OpenMP Applications*, 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 13 - 22, 2003