Hardware-Counter Based Automatic Performance Analysis of Parallel Programs *

Felix Wolf[a], Bernd Mohr[b]

[a]University of Tennessee
Innovative Computing Laboratory
Knoxville, TN 37996, USA
fwolf@cs.utk.edu

[b]Forschungszentrum Jülich
Zentralinstitut für Angewandte Mathematik
52425 Jülich, Germany
b.mohr@fz-juelich.de

The KOJAK performance-analysis environment has been designed to identify a large number of performance problems on parallel computers with SMP nodes. The current version concentrates on parallelism-related performance problems that arise from an inefficient usage of the parallel programming interfaces MPI and OpenMP while ignoring individual CPU performance. The article describes an extended design of KOJAK capable of diagnosing low individual-CPU performance based on hardware-counter information and of integrating the results with those of the parallelism-centered analysis.

1. Introduction

The performance of parallel applications is determined by a variety of different factors. Performance of single components frequently influence the overall behavior in unexpected ways. Application programmers on current parallel machines have to deal with numerous performance-critical aspects: different modes of parallel execution, such as message passing, multi-threading or even a combination of the two, and performance on individual CPUs that is determined by the interaction of different functional units. In particular, as the gap between microprocessor and memory speed increases, the understanding of processor-memory interaction becomes crucial to many optimization tasks. As a consequence, advanced performance tools are needed that integrate all these aspects in a single view.

The KOJAK performance-analysis environment has been designed to identify a large number of performance problems on typical parallel computers with SMP nodes. Performance problems are specified in terms of execution patterns to be automatically recognized in event traces. The detected patterns are then classified and quantified by type and severity, respectively. The results are presented to the user in a single integrated view along three interconnected dimensions: class of performance behavior, call path, and thread of execution. Each dimension is arranged in a hierarchy, so that the user can investigate the behavior on varying levels of detail.

While KOJAK provides a well-integrated tool to analyze parallelism-related performance problems coming from an inefficient usage of the parallel programming interfaces MPI and OpenMP, it still lacks the ability to investigate CPU and memory performance in more detail. Hardware counters integrated in modern microprocessors are an essential tool for monitoring this aspect of performance behavior. These counters exist as a small set of registers that count occurrences of specific signals related to the processor's function. Monitoring these counters facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. The article describes an extended design and implementation of KOJAK capable of diagnosing low application performance based on this type of performance data and shows how the extensions are integrated with the parallelism-centered analysis.

*This work was supported in part by the U.S. Department of Energy under Grants DoE DE-FG02-01ER25510 and DoE DE-FC02-01ER25490 and is embedded in the European IST working group APART under Contract No. IST-2000-28077

After presenting related work in Section 2, we will describe KOJAK's overall architecture and the underlying approach in Section 3. Section 4 will outline the extensions done to integrate the new type of performance problems. Finally, Section 5 will discuss limitations of the approach and propose possible enhancements.

2. Related Work

In the past few years, much research has been done on performance analysis with hardware counters. Many tool builders use libraries, such as PAPI [1] or PCL [2] which provide a standard application programming interface for accessing hardware performance counters on most modern microprocessors. Higher-level instrumentation tools, such as Dynaprof [3], CATCH [4], SCALEA [5], SvPablo [6] and TAU [7], already provide a mapping of hardware-counter information onto static or dynamic program entities. Also, the event-trace visualization tools VAMPIR [8] and Paraver [9] provide hardware-counter information as part of their time-line views.

There are also other automatic end-user tools that use hardware counters to analyze applications. Paradyn [10] was the first automatic performance tool based on a hierarchical decomposition of the search space. It searches for performance problems along various program-resource hierarchies including the call graph. Performance problems are expressed in terms of a threshold and one or more metrics, such as CPU time or message rates. The latest release uses PAPI to find memory bottlenecks. Aksum [11] uses hardware counters in its multi-experiment analysis. A distinctive feature of KOJAK in contrast to Paradyn and Aksum is the uniform mapping of all performance behavior onto execution time which allows the convenient correlation of different behavior in a single view.

Also, Fürlinger et al. [12] propose a design for an automatic online-analysis tool targeting clustered SMP architectures. On the lowest level, the tool records hardware-counter information in a distributed fashion, which is passed on to a hierarchy of agents that transform the low-level information stepwise into higher-level information. However, the design is still too early to draw a qualified comparison.

3. Performance Analysis with KOJAK

The KOJAK performance-analysis environment is depicted in Figure 1. It shows the different components with their corresponding inputs and outputs. The arrows illustrate the whole performance-analysis process from instrumentation to result presentation. The KOJAK analysis process is composed of two parts: a semi-automatic instrumentation of the user application followed by an automatic analysis of the generated performance data. KOJAK's instrumentation software runs on most major UNIX platforms and works on multiple levels, including source-code, compiler, and linker.

Instrumentation of user functions is done either during compilation using a compiler-supplied profiling interface, on the source-code level using TAU [7], or on the binary level using DPCL [13,14]. Events related to MPI are captured using a PMPI wrapper library; OpenMP constructs can be instrumented using OPARI [15] or DPOMP [14]. To get access to hardware counters the application must be linked to the PAPI (Performance API) library [1]. A more detailed de-

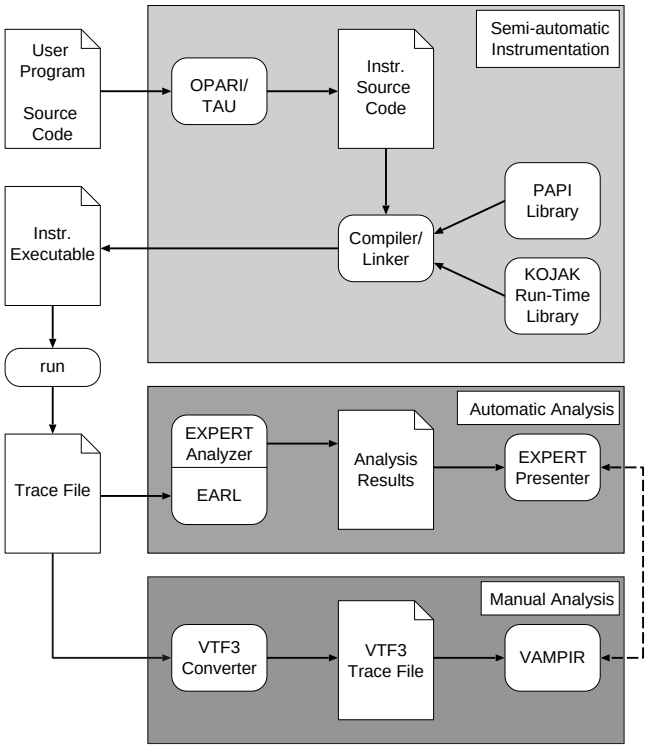

Figure 1: KOJAK overall architecture.

scription of the instrumentation process can be found in [16]. Running the resulting executable after linking generates a trace file in the EPILOG format. The EPILOG format is suitable to represent the executions of MPI, OpenMP, or hybrid parallel applications distributed across one or more (possibly large) coupled SMP systems. In addition to coupled SMPs, target systems also can be meta-computing environments as well as more traditional non-coupled or non-SMP systems.

After program termination, the trace file is fed into the EXPERT analyzer. The analyzer does not read the raw trace file as generated by the tracing library. Instead, it accesses the events through the EARL abstraction layer [16]. EARL is a high-level interface to access an event trace. Events are identified by their relative position and are delivered as a list of key-value pairs. These pairs represent event attributes, such as time and location. In addition to providing random access to single events, EARL simplifies analysis by establishing links between related events and identifying events that describe an application's execution state. The abstraction layer provides both a Python and a C++ interface and can be used independently of KOJAK for a large variety of performance-analysis tasks.

The highly integrated view of performance behavior KOJAK offers to the user is achieved by uniformly quantifying all behavior in terms of the execution time. The entire performance space is represented as a mapping of a performance problem, a call path, and a location (i.e., process or thread) onto the fraction of time spent on the problem by that particular thread in that particular call path. This time is called the severity of the tuple (problem, call path, thread). Each of the three dimensions is arranged in a hierarchy: the performance problems in a hierarchy of general and specific ones, the call tree in its natural hierarchy, and the locations in an aggregation hierarchy consisting of the levels machine, SMP node, process, and finally thread. After completion, the analyzer generates an analysis report, which serves as input for the EXPERT presenter (Figure 2). The presenter allows the user to conveniently navigate through the entire search space along all of its dimensions.

The automatic analysis can be combined with VAMPIR [8], which allows the user to investigate the patterns identified by KOJAK manually in a time-line display. To do this, the user only needs to convert the EPILOG trace file into the VTF3 format.

4. Covering Individual CPU Performance

The integration of individual-CPU performance affected all levels of the tool environment. The following subsections briefly explain the necessary changes and extensions.

4.1. Trace Format and Library

The EPILOG trace format has been extended to accommodate hardware-counter values and other system metrics, such as memory utilization[2], as part of all region-entry or exit records. A metric-description record can be used to define a system metric. The metric is assigned a name, a description, and a data type (i.e., float or integer). In addition, the user can specify whether the metric is a counter or a sample that applies to an interval or a distinct point in time, respectively. If it refers to an interval, the user can specify whether the interval starts at program start or whether the value covers only the period from the last measurement or to the next measurement. The number and order of metric-description records defines the layout of a metric-value array attached to the entry and exit records. The solution provides a high degree of flexibility, since it is not restricted to a particular set of metrics. However, to improve tool interoperability, EPILOG defines a list of names for common hardware counters with well-defined semantics. Each metric value adds eight bytes to each enter and exit record, which is one half or two thirds of their original length, respectively. Please note that all other event records remain unchanged.

To implement the new features of the trace format, a module to access hardware counters has been added to the EPILOG tracing library. The current implementation uses PAPI for the low-level access. However, the flexible module interface allows to easily integrate other hardware counter interfaces in future versions of our instrumentation system. The user specifies the desired counters via an environment variable as a colon-separated list of predefined counter names. The module achieves thread safety by creating per-thread event sets. The module interface consists mainly of three functions:

[2]Thanks to Holger Brunst for his helpful suggestions.

```
elg_metric_open();
event_set = elg_metric_create();
elg_metric_accum(event_set, value_array);
```

To initialize the module and check for availability of the requested event set, EPILOG uses the open call. After that, the create call is used to create a per-thread event set. The object returned is then supplied to all accumulate calls to read the counters for a particular thread.

4.2. Abstraction Layer

The counters or system metrics are integrated in the EARL abstraction layer as additional attributes of enter and exit events and are conveniently accessible using their name as a key. In addition, the trace-file object provides methods to query the number and kind of hardware metrics available in the trace file. The following Python code example shows how to access a metric value of an event.

```
n = trace.get_nmets()          # get number of metrics
mobj = trace.get_met(i)        # get metric i
mname = mobj['name']           # get name attribute of metric i
e = trace.event(k)             # get k-th event
print e[mname]                 # print value of metric i for event k
```

4.3. Analyzer

As pointed out in Section 3, the highly integrated view of performance behavior provided by KOJAK comes from the fact that all behavior is uniformly mapped onto execution time. However, in view of highly optimized processor architectures capable of out-of-order execution, it is hard to determine the time penalty introduced by, for example, cache misses. Even estimates are often inaccurate and highly platform dependent.

Instead, KOJAK identifies tuples (call path, thread) whose occurrence ratio of a particular hardware event is below or above a certain threshold. The execution time of these tuples delivers an upper bound of the problem's real penalty, allows the user to compare CPU-performance problems to parallelism-related problems, and narrows attention down to the affected call path and thread. We specified two experimental performance problems to be used in the analyzer.

L1 data cache misses per time above average: It computes for every tuple (call path, thread) the cache misses per time. It also computes the total number of misses and divides it by the total execution time to obtain the average miss rate. Then, the analyzer assigns to all tuples whose miss rate is above the average a severity value that is equal to the entire execution time associated with the tuple. The severity of the remaining ones is set to zero.

Floating-point operations per time below 25 % peak: This property computes for every tuple (call path, thread) the number of floating point operations per time. Since there is no way of automatically determining the peak flop rate, it must be set at installation time. Then, the analyzer assigns to all tuples whose flop rate is 25 % below the peak a severity value that is equal to the entire execution time associated with the tuple. The severity of the remaining ones is again set to zero.

The way how data is displayed by KOJAK requires siblings in the problem hierarchy to be non-overlapping. That means that a thread cannot contribute to two sibling problems during overlapping wall-clock intervals. However, inefficient cache behavior can easily coexist with weak floating-point performance or parallelism-related problems. Since the severity assigned by KOJAK to a tuple (call path, thread) with respect to a counter-related performance problem is always zero or equal to the entire execution time of the tuple, the above requirement could be relaxed to accommodate cache performance on the same level as floating-point performance.

On some platforms, not all counters needed to analyze these performance problems can be recorded simultaneously. Also, to limit the trace-file size the user might not want to record all of them in the same run. For this reason, the analyzer automatically checks for availability of certain metrics and ignores the corresponding performance problems if the data are not available.

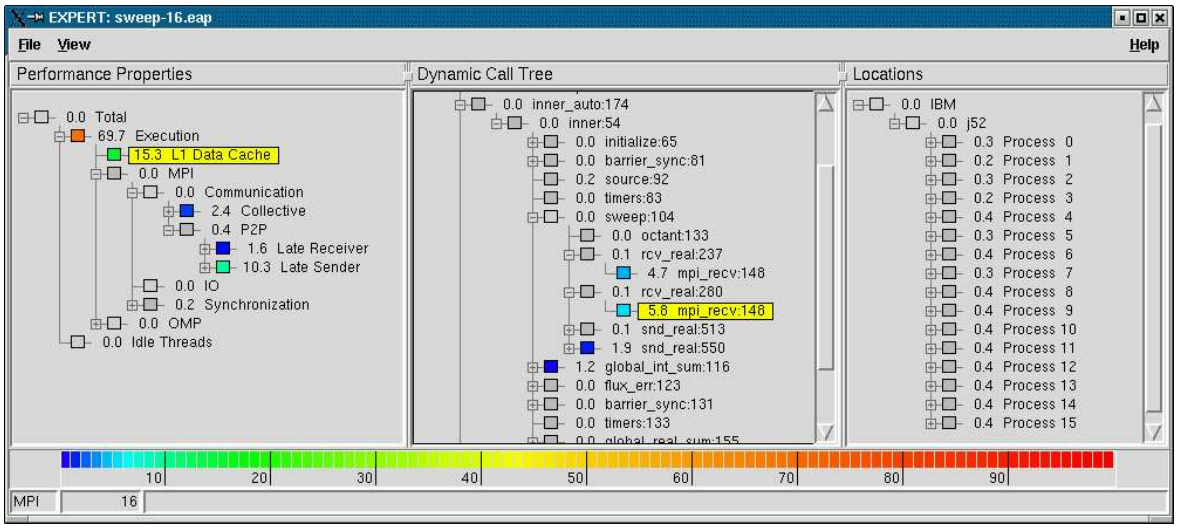

Figure 2. KOJAK's display of cache behavior for the ASCI benchmark SWEEP3D.

Figure 2 shows the cache behavior of the SWEEP3D ASCI benchmark [17] on a Power4 platform. Since PAPI cannot measure floating-point instructions and L1 data cache misses simultaneously on Power4, the analysis was restricted to L1 data cache misses only. The left pane represents the problem (i.e., property) hierarchy, the middle pane represents the call tree, and the right pane represents the location hierarchy, consisting of the levels machine, SMP node, and process. Since SWEEP3D is a pure MPI application, the thread level is hidden.

Each node in the display carries a severity value, which is a percentages of the total execution time. The value appears twice as a number and as a colored icon left to it allowing the easy identification of hot spots. The translation of colors to numbers is defined in the color scale at the bottom. By expanding or collapsing nodes in each of the three trees, the analysis can performed on different levels of granularity. A collapsed node always represents the entire subtree, an expanded node only represents itself not including its children.

The left tree shows how much time was spent on a particular performance problem by the entire program, that is across all tuples (call path, thread). In the example, the execution-time fraction of all tuples with above-average *L1 Data Cache* miss rate is 15.3 %. Selecting this performance problem, as shown in the figure, causes the middle pane to display its distribution across the call tree, whose nodes are labeled with a function name together with the line number from which it was called. Apparently, some of the MPI calls exhibit a miss rate above the average, whereas the computational parts seem to be without major findings. Finally, the right tree shows the severity of the selected call path broken down to different processes.

Interesting is that among the identified call paths the two with the highest execution time (i.e., the selected one with 5.8 % and another one with 4,7 %) have also been identified to be the source of a non-negligible *Late Sender* problem, that is, nearly all of the execution time was actually spent waiting on a message to be received (not shown in the figure). Therefore, the question arises whether a more cache-friendly receive function would be able to significantly speed up the application, since it wouldn't necessarily speed up the delivery of the message it was waiting for most of the time.

5. Conclusion

As the previous example suggests, KOJAK's ability to analyze parallelism and individual-CPU performance problems simultaneously can provide useful insights into the performance behavior of a parallel application and help avoid hastily conclusions that might occur based on lesser integrated data.

However, while the way KOJAK defines the severity of counter-based performance problems allows a high level of integration with parallelism-related behavior, it does not give much information on the actual run-time penalty. Also, the classification 'above or below a certain threshold' does not even

say how far above or below. Therefore, we believe that the time-based view should be combined with additional views that provide the actual occurrence numbers and rates of the hardware events. Also, the current set of counter-based problems (i.e., cache and floating point) needs to be extended to cover additional aspects of individual-CPU performance, such as, for example, TLB misses. Since often not all desired counters can be recorded simultaneously, methods are needed to combine the data obtained from multiple experiments offline - thereby taking into account the limited reproducibility of a single experiment.

REFERENCES

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, A Portable Programming Interface for Performance Evaluation on Modern Processors, The International Journal of High Performance Computing Applications 14 (3) (2000) 189–204.

[2] R. Berrendorf, H. Ziegler, PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors, Tech. Rep. FZJ-ZAM-IB-9816, Forschungszentrum Jülich (October 1998).

[3] P. Mucci, Dynaprof home page, `http://www.cs.utk.edu/~mucci/dynaprof/`.

[4] L. A. DeRose, F. Wolf, CATCH – A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications, in: Proc. of the 8th International Euro-Par Conference, Lecture Notes in Computer Science, Paderborn, Germany, 2002, pp. 167–176.

[5] H.-L. Truong, T. F. G. Madsen, A. D. Malony, H. Moritsch, S. Shende, On Using SCALEA for Performance Analysis of Distributed and Parallel Programs, in: Proc. of the Conference on Supercomputers (SC2000), Denver, Colorado, 2001.

[6] L. DeRose, D. A. Reed, SvPablo: A Multi-Language Architecture-Independent Performance Analysis System, in: Proc. of the International Conference on Parallel Processing (ICPP'99), Fukushima, Japan, 1999.

[7] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, S. Karmesin, Portable Profiling and Tracing for Parallel Scientific Applications using C++, in: Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools, ACM, 1998, pp. 134–145.

[8] A. Arnold, U. Detert, W. E. Nagel, Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding, in: R. Winget, K. Winget (Eds.), Proc. of Cray User Group Meeting, Denver, CO, 1995, pp. 252–258.

[9] European Center for Parallelism of Barcelona (CEPBA), Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual, `http://www.cepba.upc.es/paraver/` (November 2000).

[10] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, T. Newhall, The Paradyn Parallel Performance Measurement Tool, IEEE Computer 28 (11) (1995) 37–46.

[11] C. Seragiotto Júnior, M. Geissler, G. Madsen, H. Moritsch, On Using Aksum for Semi-Automatically Searching of Performance Problems in Parallel and Distributed Programs, in: Proc. of 11th Euromicro Conf. on Parallel Distributed and Network based Processing (PDP 2003), Genua, Italy, 2003.

[12] K. Fürlinger, M. Gerndt, Distributed Application Monitoring for Clustered SMP Architectures, in: Proc. of the 9th International Euro-Par Conference, Klagenfurt, Austria, 2003.

[13] L. DeRose, T. Hoover Jr., J. K. Hollingsworth, The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools.

[14] L. DeRose, B. Mohr, S. Seelam, An Implementation of the POMP Performance Monitoring Interface for OpenMP Based on Dynamic Probes, in: Proc. of the 5th European Workshop on OpenMP (EWOMP'03), Aachen, Germany, 2003.

[15] B. Mohr, A. Malony, S. Shende, F. Wolf, Design and Prototype of a Performance Tool Interface for OpenMP, The Journal of Supercomputing 23 (2002) 105–128.

[16] F. Wolf, Automatic Performance Analysis on Parallel Computers with SMP Nodes, Ph.D. thesis, RWTH Aachen, Forschungszentrum Jülich, ISBN 3-00-010003-2 (February 2003).

[17] Accelerated Strategic Computing Initiative (ASCI), The ASCI sweep3d Benchmark Code, `http://www.llnl.gov/asci_benchmarks/`.