Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0

Kevin A. Huck[*], Allen D. Malony, Sameer Shende and Alan Morris

Performance Research Laboratory
Computer and Information Science Department
University of Oregon
1201 University of Oregon, Eugene, OR, 97413 USA
khuck@cs.uoregon.edu
Phone: (541)346-4409, Fax: (541) 346-5373

Abstract

The integration of scalable performance analysis in parallel development tools is difficult. The potential size of data sets and the need to compare results from multiple experiments presents a challenge to manage and process the information. Simply to characterize the performance of parallel applications running on potentially hundreds of thousands of processor cores requires new scalable analysis techniques. Furthermore, many exploratory analysis processes are repeatable and could be automated, but are now implemented as manual procedures. In this paper, we will discuss the current version of PerfExplorer, a performance analysis framework which provides dimension reduction, clustering and correlation analysis of individual trails of large dimensions, and can perform relative performance analysis between multiple application executions. PerfExplorer analysis processes can be captured in the form of Python scripts, automating what would otherwise be time-consuming tasks. We will give examples of large-scale analysis results, and discuss the future development of the framework, including the encoding and processing of expert performance rules, and the increasing use of performance metadata.

Keywords: parallel performance analysis, data mining, scalability, scripting, metadata, knowledge supported analysis

[*]Corresponding author

1 Introduction

Parallel applications running on high-end computer systems manifest a complex combination of performance phenomena, such as communication patterns, work distributions, and parametric study results. Tools that analyze parallel performance attempt to observe these phenomena in measurement datasets captured by instrumentation of the source code with timers, or by periodically sampling the program counter during runtime. The resulting datasets are rich with information, potentially relating multiple performance metrics to performance variations and parameters specific to the application-system experiment.

One common representation of performance data is as performance profiles. Each profile represents an aggregation of one metric as measured in one region of code on one thread of execution, such as how many floating point operations were executed, or how many cache misses occurred. Analysis tools use this data to identify behavior such as performance hot spots and irregular work distribution across threads of execution. They can also calculate and display summary statistics, such as average time (across threads) spent in one function, or correlate between the time spent in two or more functions. While the results from current performance tools are useful to trained analysis experts, next generation tools need to go beyond the display of summary statistics and provide in-depth analysis and explanation of performance results to the user.

The TAU Performance System [25] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, and Python. Instrumentation and measurement tools such as TAU can collect very detailed performance data from parallel applications. The potential sizes of datasets and the need to assimilate results from multiple experiments makes it a challenge to both process the information and discover and understand new insights about performance. In order to perform analysis on these large collections of performance experiment data, we developed PerfExplorer [9], a framework for parallel performance data mining and knowledge discovery. The framework architecture enables the development and integration of data mining operations that can be applied to parallel performance profiles. PerfExplorer is built on a performance data management framework called PerfDMF [8], which provides a library to access the parallel profiles and save analysis results in a relational database. PerfDMF includes support for nearly a dozen performance profile formats, including TAU profiles. The application is integrated with existing analysis toolkits (R [29], Weka [32]), and provides for extensions using those toolkits. Both PerfDMF and PerfExplorer are free, open-source tools included in the TAU distribution.

A performance data mining framework should support both advanced analysis techniques as well as extensible *meta analysis* of performance results. The use of *process control* for analysis scripting (see Section 2.1), *persistence* and *provenance* mechanisms for retaining analysis results and history (see Section 2.4), *metadata* for encoding experiment context (see Section 2.2), and support for *reasoning* about relationships between performance characteristics and behavior (see Section 2.3) all are important for productive performance analytics. The framework must also be concerned about how to interface with application developers in the performance discovery process. The ability to engage in process programming (the ability to capture analysis workflows), knowledge engineering (including the performance context and inference rules which explain performance results), and results management opens the framework tool set for creating data mining environments specific to the developer's concerns.

We have redesigned our integrated framework for performing meta analysis to incorporate parallel performance data, performance context metadata, expert knowledge, and intermediate analysis results. Methods were required for correlating context metadata with the performance data and the analysis results in order to provide the capability to generate desired empirical performance results from accurate suggestions on how to improve performance. Constructing this framework also required methods for encoding expert knowledge to be included in the analysis of performance data from parametric experiments. Knowledge about subjects such as hardware configurations, libraries, components, input data, algorithmic choices, runtime configurations, compiler choices, and code changes will augment direct performance measurements to make additional analysis combinations possible.

The remainder of the article is as follows. We discuss our analysis approach for the framework and our implementation in Section 2. We will present some recent analysis examples which demonstrate some new PerfExplorer features in Section 3, discuss related work in Section 4 and present future work and concluding remarks in Section 5.

2 PerfExplorer Design

PerfExplorer[8] was originally designed as a Java application for data mining multi-experiment parallel performance profiles. Its capabilities included general statistical analysis of performance data, dimension reduction, clustering, and correlation of performance data, and multi-experiment data query and management. These functions were provided by existing analysis toolkits (R[29] and Weka[32]), and our profile database system PerfDMF[8].

While PerfExplorer was a step forward in the ability to automatically process complex statistical functions on large amounts of multi-dimensional parallel performance data, its functionality was limited in two respects. First, the tool only allowed a user to select from a limited number of analysis operations via a graphical user interface. Reliably repeatable and user configurable analysis processes were not possible. Second, PerfExplorer only provided new visualizations and descriptions of the data – it did not *explain* the performance characteristics or behavior observed (i.e., meta analysis). Scripting and support for retaining intermediate results helped to address the first shortcoming. The second was more challenging.

For example, an analyst can determine that in a test using 16 processors, application X spent 30% of its total execution time in function `foo()`, and that when the number of processors is increased to 32, the percentage of time may go up, down, or stay the same, depending on the purpose of the function. However, PerfExplorer did not have the capability to explain *why* the change happened. The explanation may be as simple as the fact that the input problem also doubled in size, but without that contextual knowledge, no analysis tool could be expected to come to any conclusions about the cause of the performance change without resulting to speculation.

As we discuss our enhancements to PerfExplorer, we will consider two analysis cases: 1) we have collected parallel performance data from multiple experiments, and we wish to compare their performance, or 2) we have collected performance data from one experiment, and would like to compare the performance between processes or threads of execution. Like other tools, PerfExplorer could provide the means for an analyst to determine which execution is the "best" and which is the "worst", and can even help the analyst investigate further into which regions of code are most affected, and due to which metrics. However, there was no explicit *process control*, which is required in order to perform repeated analysis procedures or non-interactive analysis automation, nor was there *higher-level reasoning* or *analysis* of the performance result in order to explain what may have caused the performance differences. In order to perform these types of meta-analysis, several components are necessary to meet the desired goals.

Figure 1 shows the redesigned PerfExplorer components, and Figure 2 shows the interaction between components in the new PerfExplorer design. The performance data and accompanying metadata, discussed in Section 2.2, are stored in the PerfDMF database. Performance data is used as input for statistical analysis and data mining operations, as was the case in the original version of PerfExplorer. The new design adds the ability to make all intermediate analysis data and final results persistent. Expert knowledge is incorporated into the analysis, and these new inputs allow for higher-level analysis. The expert knowledge can be application specific, machine specific, or just general parallel computing knowledge. An inference engine is added to combine the performance data, analysis results, expert knowledge and execution metadata into a performance characterization. The provenance of the analysis result is stored with the result, along with all intermediary data, using object persistence. The whole process is contained within a process control framework, which provides user control over the performance characterization process.

2.1 Process Control

One of the key aspects of the new PerfExplorer design is the requirement for process control. While user interfaces and data visualization are useful for interactive data exploration, the user will need the ability to control the analysis process as a discrete set of operations.

There are several types of parametric study commonly seen in the parallel performance literature: application benchmarking, machine benchmarking, application performance testing and workload characterization. For each of these studies, application performance data is collected while varying one or more configuration parameters. Usually, the data collection process is automated, to prevent errors or omissions. In each of these studies, the analysis process can and should be automated in order to prevent analysis mistakes and to streamline the analysis processing.

In order to chain analysis operations together in a repeatable framework, PerfExplorer required an extension mechanism for creating higher-order analysis procedures. One way of doing this is through a scripting interface, such as Jython[20], a full Python interpreter written in Java. Because PerfExplorer is a Java application, all of the application objects are available to the script interface, but we limit the access to a smaller subset API. With the interface, it is straightforward to derive new metrics, perform analysis, and automate the processing of performance data. An example script is shown in Figure 3. This simple example loads some general purpose inference rules, loads a trial from PerfDMF, derives floating point operations per second, and then compares each event's exclusive value with the inclusive value of main before processing the rules, where an event is defined as any instrumented code region.

2.2 Collecting and Integrating Metadata

Performance instrumentation and measurement tools such as TAU[25] collect context metadata along with the application performance data. This metadata contains potentially useful information about the build environment, runtime environment, configuration settings, input and output data, and hardware configuration. Metadata examples which are automatically collected by the profiling provided by TAU include fields such as processor speed, node hostname, and cache size. By integrating these fields into the analysis process, the analysis framework can reason about potential causes for performance failures.

The TAU instrumentation and measurement toolkit provides three ways to acquire metadata for analysis:

- The default behavior for the TAU measurement toolkit is to collect common hardware and software metadata from the runtime environment, such as processor speed, memory size, cache size, operating system version, etc. Table 1 shows examples of metadata fields which are automatically collected by the profiling provided by TAU. It should be easy to see how fields such as *CPU MHz*, *Cache Size* or *Memory Size* would be useful in explaining the differences between executions. In addition, on specialized hardware such as the IBM BlueGene/L or BlueGene/P systems, there are additional system calls which can provide detailed information about the hardware and the logical mapping of the processes to physical nodes.

- The TAU instrumentation API has a method, `TAU_METADATA()`, which the application analyst can insert into the code. This is the primary way for an end user to collect metadata about their application. The method takes two parameters, a name and a value. Any data of interest can be inserted into the metadata to be used later in the analysis. Input variables, runtime configuration settings, application arguments, and domain decompositions can be specified by the user.

- The PerfDMF data importer can take an optional XML file with metadata fields which contain name/value pairs to be included in the performance metadata. The schema is very simple, and does not require special XML processing libraries to generate. Information relating to the build environment, compiler options, input files, batch system, allocated hardware, or anything else that might assist the performance analysis can be included in this XML file.

2.3 Inference Engine

In order to provide the type of higher-level reasoning and meta-analysis we require in our design, we have integrated a JSR-94 [27] compliant rule engine, JBoss Rules[21]. The selection of an inference engine and processing rules allows another method of flexible control of the process, and also provides the possibility of developing a domain specific language to express the analysis.

As mentioned in Section 2.1, there are several types of parametric study commonly seen in the parallel performance literature. In the example of a scalability study, the number of processors used and the input problem size is varied, and empirical performance results are compared with expected results, based on baseline comparisons. In each of these parametric studies, we have identified eight common categories of parameters, listed in Table 2, along with example parameters for each category and an example of a known assumption, or *expert knowledge*, about a parameter in that category that could be helpful in analyzing the performance of an experiment.

As an example, the first category includes differences between architectures, such as when porting an application, or performing an application benchmarking study on more than one architecture. Parameters such as CPU type and speed, the amount of cores per CPU, the number of CPUs per node, etc. all represent useful information when comparing two or more architectures. In order to utilize this information, performance assumptions can be made in the analysis process which will help guide the analysis. For example, consider an application executed with the same configuration on two different machines. If the metadata shows that the only difference between the two machines is the speed of the CPU, then the analysis should correlate the performance differences between the two executions to the differences in speed. As another example, suppose that we can identify a region of code as inherently sequential. Any scalability analysis of this region could then assume that there will be no expected improvement by increasing the number of processors, and will not flag this section as a performance bottleneck. While these are overly simplified examples, they illustrate the potential utility that expert knowledge about an execution can provide to the performance analysis. Some expert knowledge would be specific to the analysis task at hand, while other examples would be reusable across many if not all parametric studies.

An example rule is shown in Figure 4. This example rule will fire for any and all events which have a lower than average L2 cache hit rate, and also account for at least 10% of the total run time. In this example, the conclusion is output to the user, but the rules can also fire other scripts, or request operations from the PerfExplorer API directly. Facts which result in the execution of rules can be asserted in PerfExplorer operations directly, or facts can be asserted by the scripts.

2.4 Provenance and Data Persistence

In order to rationalize analysis decisions, any explanation needs to include the data provenance, or the full chain of evidence and handling from raw data to synthesized analysis result. The new PerfExplorer design will include the ability to make all intermediate analysis data persistent, not just the final summarization. The provenance of the analysis result is stored with the results and all intermediary data, using object persistence[22]. Any scientific endeavor is considered to be of "good provenance" when it is adequately documented in order to allow reproducibility. For parallel performance analysis, this includes all raw data, analysis methods and parameters, intermediate results, and inferred conclusions.

3 Analysis Examples

3.1 S3D

S3D[2] is a multi-institution collaborative effort with the goal of creating a terascale parallel implementation of a turbulent reacting flow solver. S3D uses direct numerical simulation (DNS) to model combustion science which produces high-fidelity observations of the micro-physics found in turbulent reacting flows as well as the reduced model descriptions needed in macro-scale simulations of engineering-level systems. The examples described here were run on Jaguar[19], the hybrid Cray XT3/XT4 system at Oak Ridge National Laboratory (ORNL).

During scalability tests (from 1 to 12,000 processors) of S3D instrumented with TAU, it was observed that as the number of processors exceeded 1728, the amount of time spent in communication began to grow significantly, and `MPI_Wait()` in particular composed a significant portion of the overall run time (approximately 20%). By clustering the performance data in PerfExplorer, it was observed that there were two natural clusters in the data. The first cluster consisted of a majority of the processes, and these nodes spent less time in main computation loops, but a long time in `MPI_Wait()`. The other cluster of processes spent slightly more time in main computation loops, and far less time in `MPI_Wait()`.

By automatically collecting the MPI host names with the TAU metadata collection, we were able to determine, at runtime, the names of the nodes on which the processes ran. The node IDs were stored in the metadata with the performance data. In the case of a 6400 process run, as shown in Figure 5, there were again two clusters, with 228 processes in one cluster having very low `MPI_Wait()` times (about 40 seconds), and the remainder of the processes in one cluster having very high `MPI_Wait()` times (over 400 seconds). The metadata was then manually correlated with information about the hardware characteristics of each node, identified the slower nodes as XT3 nodes, and the faster nodes as XT4 nodes. There are two primary differences between the XT3 and XT4 partitions. The XT3 nodes have slower DDR-400 memory (5986 MB/s) than the XT4 nodes' DDR2-667 memory (7147 MB/s), and the XT3 partition has a slower interconnection network (1109 MB/s v. 2022 MB/s). Because the application is memory intensive, the slower memory modules have a greater effect on the overall runtime, causing the XT3 nodes to take longer to process, and subsequently causing the XT4 nodes to spend more time in `MPI_Wait()`.

In order to remove this last manual step to correlating application performance with hardware characteristics, we needed more information about the nodes than was available from the metadata. By using the `nodeinfo` utility available from the batch system, we were able to collect information about each node in the allocation, including the memory speed and interconnect speed, which directly identify the XT3 and XT4 nodes in the full machine. Using a python script, the `nodeinfo` data was formatted as XML, and loaded with the performance data using the third method outlined in Section 2.2. A PerfExplorer script was written which loaded the trial data, extracted the five most time consuming code regions and correlated the event performance with the metadata fields for each thread of execution. An inference rule was used to identify the code regions which had an effectively inverse correlation between run time and both memory speed and interconnect speed.

Running S3D on an XT4-only configuration yielded roughly a 12% time to solution reduction over the hybrid configuration, primarily by reducing `MPI_Wait()` times from an average of 390 seconds down to 104 seconds. If this application is to be run on a heterogeneous configuration of this machine or any other, load balancing should be integrated which takes into consideration the computational capacity of each respective processor. The use of metadata would also be important for this optimization.

3.2 GTC

The Gyrokinetic Toroidal Code (GTC)[5] is a particle-in-cell physics simulation which has the primary goal of modeling the turbulence between particles in the high energy plasma of a fusion reactor. Scalability studies of the original large-scale parallel implementation of GTC (there are now a small number of parallel implementations, as the development has fragmented) show that the application scales very well - in fact, it scales at a better than linear rate. However, discussions with the application developers revealed that it had been observed that the application gradually slows down as it executes[31] - each successive iteration of the simulation takes more time than the previous iteration.

In order to measure this behavior, the application was auto-instrumented with TAU, and manual instrumentation was added to the main iteration loop to capture dynamic phase information. The application was executed on 64 processors of the Cray XT3/XT4 system at ORNL for 100 iterations, and the performance data was loaded into PerfDMF. A simple analysis script was constructed in order to examine the dynamic phases in the execution. The script was used to load the performance data, extract the dynamic phases from the profile, calculate derived metrics (L1 and L2 cache hit ratios, FLOPs), calculate basic statistics for each phase, and graph the resulting data as a time series showing average, minimum and maximum values for each iteration.

As shown in Figure 6, during a 100 iteration simulation, each successive iteration takes slightly more time than the previous iteration. Over the course of the test simulation, the last iteration takes nearly one second longer than the first iteration. As a minor observation, every fourth iteration results in a significant increase in execution time. Hardware counters revealed that the L2 cache hit-to-access ratio decreases from 0.92 to 0.86 (L1 cache hit-to-access ratios also decrease, but to a lesser extent). Subsequently, the GFLOPs per processor rate decreases from 1.120 to 0.979. Further analysis of the routines called from the main loop show that the decrease in execution is limited to two routines, `CHARGEI` and `PUSHI`. In the `CHARGEI` routine, each particle in a region of the physical subdomain applies a charge to up to four cells, and in the `PUSHI` routines, the particle locations are updated by the respective cells after the forces are calculated. The increase in time every fourth iteration is due to a diagnostic call, which happens every `ndiag` iterations, an input parameter captured as metadata.

A second script for this problem was also developed, which loaded the performance data, extracted the top ten time consuming code regions, derived the L1, L2 and FLOPs metrics, and then compared each code region to the overall performance of the application. An inference rule was constructed which identified code regions which had lower than average cache hit ratios. The combination of this script and rule identified the same code functions, `CHARGEI` and `PUSHI`, as having poor cache behavior. Another script possibility, which was not explored, would be to correlate the iteration number with the performance of each code region. We would expect to see that the reduced cache hit ratios would be identified as correlated with the iteration number.

Discussions with other performance analysis experts on the project revealed that the `CHARGEI` and `PUSHI` routines have good spatial locality when referencing the particles, however over time, they have poor temporal locality when referencing the grid cells. As a result, access to the grid cells becomes random over time. Further analysis is necessary to determine whether the expense of re-ordering the particles at the beginning of an iteration could be amortized over a number of iterations, and whether this added cost would yield a benefit in the execution time. While it appears that the performance degradation levels out after roughly 30 iterations, it should be pointed out that a full run of this simulation is at least 10,000 iterations, and as the 5,000 iteration execution shows in Figure 6 (d), the performance continues to degrade. Assuming a 10,000 iteration execution would take an estimated 20 hours to complete on the Cray XT3/XT4, potentially 2.5 hours of computation time per processor could be saved by improving the cache hit ratios.

4 Related Work

Hercule[15, 14, 13] is a parallel performance diagnosis tool which uses the expert system CLIPS to process computational model-centric rules which can diagnose common performance problems. Hercule's rules define symptoms of known parallel application problems, such as *load imbalance*, *insufficient parallelization*, etc., and encodes possible solutions for correcting these known problems. Hercule takes as input the application's parallel model, and diagnoses known problems from the input data and the application model assumptions. Hercule analyzes event trace files, not profiles.

EXPERT[26], from the KOJAK[12] project, is an automatic event-trace analysis tool for MPI and OpenMP applications. It searches the traces for execution patterns indicating low performance and quantifies them according to their severity. The patterns target both problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance. Unlike our proposed approach, EXPERT searches for known problems, rather than focusing on characterization and new problem discovery. Also, the performance data analyzed is trace data. CUBE[33] is a graphical browser suitable for displaying a wide variety of performance measurements for parallel programs including MPI and OpenMP applications, and is the primary analysis viewer for SCALASCA[7], a parallel implementation of the EXPERT trace analysis methods. CUBE implements Performance Algebra[11], a technique for performing difference, merge and aggregation operations on parallel performance profile data. While CUBE provides a powerful interface for visualization and exploratory analysis of the differences between two performance data sets, there is no mechanism for linking the performance behavior to the performance context, and providing the user with a meaningful explanation of why the performance differs between the two profiles.

Paradyn[18] utilizes the Performance Consultant[11] and Distributed Performance Consultant[23] for run-time and offline discovery of known performance problems. The latest version of the Performance Consultant uses historical performance data to help guide bottleneck detection. While the Performance Consultant does include contextual information about the runtime environment to help explain performance differences, there doesn't appear to be a mechanism for including additional expert knowledge about the application, such as data or event relationships. And like the aforementioned tools, the Performance Consultant's strength is in diagnosing known performance problems, rather than general performance characterization.

KappaPi[3, 4] (Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement) and KappaPI2[10] are tools which use trace files from PVM and MPI applications, detect known performance bottlenecks, and determine causes by applying inference rules. The causes are then related back to the source code and suggest recommendations to the user.

Performance Assertions[30] have been developed to confirm that the empirical performance data of an application or code region meets or exceeds that of the expected performance. By using the Assertions, the programmer can relate expected performance results to variables in the application, the execution configuration (i.e. number of processors), and pre-evaluated variables (i.e. peak FLOPS for this machine). This technique allows users to encode their performance expectations for regions of code, confirm these expectations with empirical data, and even make runtime decisions about component selection based on this data. The use of performance assertions requires extensive annotation of source code, and requires the application developer's experience and intuition in knowing where to insert the assertions, and what kind of performance result to expect.

JavaPSL[6] is a Java Performance Specification Language, designed to be used to specify techniques for searching for known performance problems such as poor scaling, load imbalance, and communication overhead. The specification language could be useful in the application of search heuristics in a particular diagnosis process, and represents a good example of the type of low-level analysis whose results could be used in conjunction with expert knowledge and context metadata to suggest the causes of performance phenomena.

Directly relevant to PerfDMF are the projects that utilize a performance database as a component of a performance analysis system, particularly for multi-experiment performance analysis. The SIEVE (Spreadsheet-base Interactive Event Visualization Environment) system [24] showed the benefit of a simple table-based structuring of performance data coupled with a programmable analysis engine. More sophisticated performance data models, such as found in Paradyn [18] and CUBE [26], allow a richer analysis algebra to be applied to multi-experiment performance information.

HPCToolkit [16] is able to merge data from multiple performance experiments in a database that is correlated with the program source and hyperlinked for analysis and viewing with the HPCView [17] tool. Performance data manipulated by HPCView can come from any source, as long as the profile data can be translated or saved directly to a standard, profile-like input format. The toolkit provides a interactive interface that allows the user to define expressions to compute derived metrics as functions of the measured data and of previously-computed derived metrics.

The Prophesy system [28] successfully applies a performance database to manage multi-dimensional performance information for parallel analysis and modeling. The database is a core component of the system, implemented using relational DBMS technology and storing detailed information from the Prophesy measurement system and performance modeling processes. Prophesy also uses some statistical analysis methods to do application performance analysis and prediction.

The primary inspiration for the data mining aspect of PerfExplorer is the research by Ahn and Vetter[1]. Those authors chose to use several multivariate statistical analysis techniques to analyze parallel performance behavior. The types of analysis they performed included cluster analysis and F-ratio, factor analysis, and principal component analysis. They showed how hardware counters could be used to analyze the performance of multiprocessor parallel machines.

5 Future Work and Concluding Remarks

In this paper, we have discussed the new design and implementation of PerfExplorer, including components for scripting, metadata encoding, expert rules, provenance and data persistence. In our examples, we have discussed how features such as metadata encoding and scripting aid in the analysis process. Development of general purpose, application and machine specific inference rules is ongoing. In the future, we hope that PerfExplorer will be distributed with an extensive library of analysis scripts, and accompanying inference rules. However, the real strength of the framework is the ability for users to customize the analysis to fit the task at hand. While the metadata support in PerfExplorer allows for some manual correlation between contextual information and performance results, more extensive analysis rules to interpret the results with respect to the contextual information would aid us in our long term goal of a performance tool which would summarize performance results and link them back to the actual causes, which are essentially the context metadata relating to the application, platform, algorithm, and known related parallel performance problems. Encoding this knowledge form that our performance tool can use is instrumental in developing new analysis techniques that capture more information about the experiment than simply the raw performance data.

6 Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-07ER25826 and DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science and the NSF (grant #CCF0444475). The authors would like to thank PERI, SciDAC, ORNL, NERSC and RENCI for including us in the PERI SciDAC project, and a special thanks to John Mellor-Crummey for a better understanding of the locality issues in GTC.

References

[1] D. Ahn and J. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of Supercomputing*, 2002.

[2] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummy, N. Podhorski,

R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. (to appear) *Institute of Physics (IOP) Journal*, 2007.

[3] A. Espinosa, T. Margalef, and E. Luque. Automatic performance evaluation of parallel programs. In *IEEE Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, January 1998.

[4] A. Espinosa, T. Margalef, and E. Luque. Automatic performance analysis of PVM applications. In *EuroPVM/MPI*, volume LNCS 1908, pages 47–55, 2000.

[5] S. Ethier, W.M. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *Journal of Physics: Conference Series*, 16:1–15, 2005.

[6] Thomas Fahringer and Clóvis Seragiotto Júnior. Modeling and detecting performance problems for distributed and parallel programs with JavaPSL. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 35–35, New York, NY, USA, 2001. ACM Press.

[7] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *LNCS*, pages 303–312, Bonn, Germany, September 2006. Springer.

[8] K. Huck, A. Malony, R. Bell, and A. Morris. Design and implementation of a parallel performance data management framework. In *Proceedings of the International Conference on Parallel Computing, 2005 (ICPP2005)*, pages 473–482, 2005.

[9] Kevin A. Huck and Allen D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Conference on High Performance Networking and Computing (SC'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[10] J. Jorba, T. Margalef, and E. Luque. Performance analysis of parallel applications with kappapi2. *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, 33:155–162, 2006.

[11] K. Karavanic and B. Miller. *On-Line Monitoring Systems and Computer Tool Interoperability*, chapter A Framework for Multi-Execution Performance Timing. Nova Science Publishers, New York, USA, 2003.

[12] KOJAK. Kojak. `http://www.fz-jeulick.de/zam/kojak/`, 2006.

[13] L. Li and A. D. Malony. Model-based performance diagnosis of master-worker parallel computations. In *Europar 2006*, 2006.

[14] Li Li and Allen D. Malony. Knowledge engineering for automatic parallel performance diagnosis. *Concurrrency and Computation: Practice and Experience* to appear, 2006.

[15] Li Li, Allen D. Malony, and Kevin Huck. Model-based relative performance diagnosis of wavefront parallel computations. In *HPCC*, Munich, Germany, 2006.

[16] J. Mellor-Crummey. Hpctoolkit: Multi-platform tools for profile-based performance analysis. In *5th International Workshop on Automatic Performance Analysis (APART)*, November 2003.

[17] J. Mellor-Crummey, R. Fowler, and G. Marin. Hpcview: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.

[18] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[19] National Center for Computational Sciences. Resources - National Center for Computational Sciences (NCCS). `http://info.nccs.gov/resources/jaguar`, September 2007.

[20] Python Software Foundation. The Jython Project. `http://www.jython.org/`.

[21] Red Hat Middleware, LLC. Jboss.com - jboss rules. `http://www.jboss.com/products/rules`.

[22] Red Hat Middleware, LLC. Relational Persistence for Java and .NET. `http://www.hibernate.org/`, 2007.

[23] P. Roth. Towards automatic performance diagnosis on thousands of nodes. 5th International APART Workshop, SC2003 Conference, November 2003.

[24] S. Sarukkai and D. Gannon. Sieve: A performance debugging environment for parallel programs. *J. Parallel Distrib. Comput.*, 18(2):147–168, 1993.

[25] Sameer Shende and Allen D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.

[26] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proceedings of 2004 Interna-*

tional Conference on Parallel Processing (ICPP'04), pages 63–72, Montreal, Quebec, Canada, 2004.

[27] Sun Microsystems. The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR#94. `http://jcp.org/en/jsr/detail?id=94`, 2008.

[28] V. Taylor, X. Wu, and R. Stevens. Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.

[29] The R Foundation for Statistical Computing. R project for statistical computing. `http://www.r-project.org`, 2007.

[30] Jeffrey S. Vetter and Patrick H. Worley. Asserting performance expectations. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[31] Nathan Wichmann, Mark Adams, and Stephane Ethier. New advances in the gyrokinetic toroidal code and their impact on performance on the Cray XT series. In *Cray Users Group*, 2007.

[32] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, San Francisco, 2nd edition, 2005. `http://www.cs.waikato.ac.nz/~ml/weka/`.

[33] F. Wolf and B. Mohr. Automatic performance analysis of SMP cluster applications. Technical Report 05, Research Centre Julich, 2001.

Field	**Example**
CPU Cores	4
CPU MHz	2660.006
CPU Type	Intel(R) Xeon(R) CPU X5355 @ 2.66GHz
CPU Vendor	GenuineIntel
CWD	/home/joeuser/tau2/examples/NPB2.3/bin
Cache Size	4096 KB
Executable	/home/joeuser/tau2/examples/NPB2.3/bin/lu.C.16
Hostname	garuda.cs.uoregon.edu
Local Time	2007-03-29T16:06:08-07:00
Memory Size	8155912 kB
Node Name	garuda.cs.uoregon.edu
OS Machine	x86_64
OS Name	Linux
OS Release	2.6.18.1_ktau_1.7.9_pctr
OS Version	#2 SMP Mon Mar 26 17:36:14 PDT 2007
TAU Architecture	x86_64
TAU Config	-fortran=intel -cc=icc -c++=icpc -mpi ...
UTC Time	2007-03-29T23:06:08Z
username	joeuser

Table 1: Default TAU metadata field examples.

Category	**Parameter Examples**	**Possible Assumptions**
Machines	processor speed/type, memory size, number of cores	CPU A faster than CPU B
Components	MPI implementation, linear algebra library, runtime component	component A faster than B
Input	problem size, input data, problem decomposition	smaller problem means faster execution, vice-versa
Algorithms	FFT vs. DFT	algorithm A faster than B for problem $> X$
Configurations	number of processors, runtime parameters, number of iterations	more processors means faster execution, vice-versa
Compiler	compiler choice, compiler options, pre-compiler usage, code transformations	execution time: -O0 $\geq$ -O1 $\geq$ -O2 $\geq$ -O3 $\geq$ -fast
Code Relationships	call order, send-receive partners, concurrency, functionality	code region has expected concurrency of X
Code Changes	code change between revisions	newer code expected to be faster

Table 2: Parametric categories and corresponding example assumptions in those categories.

List of Figures

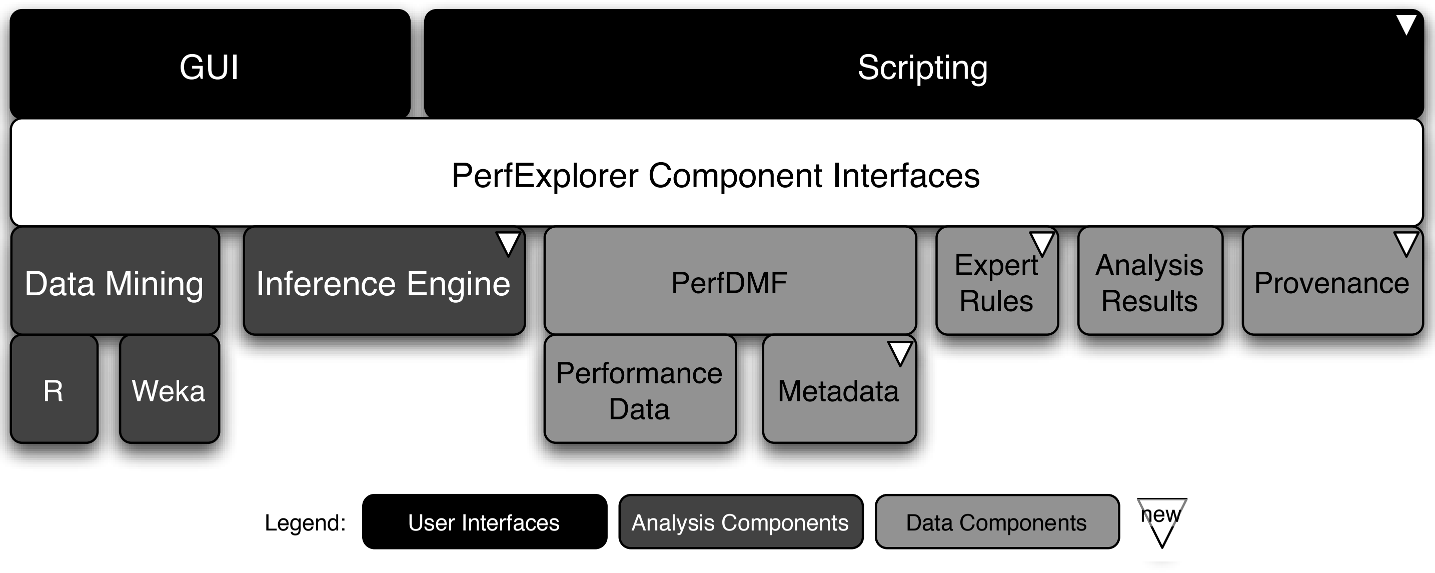

Figure 1: The redesigned PerfExplorer components.

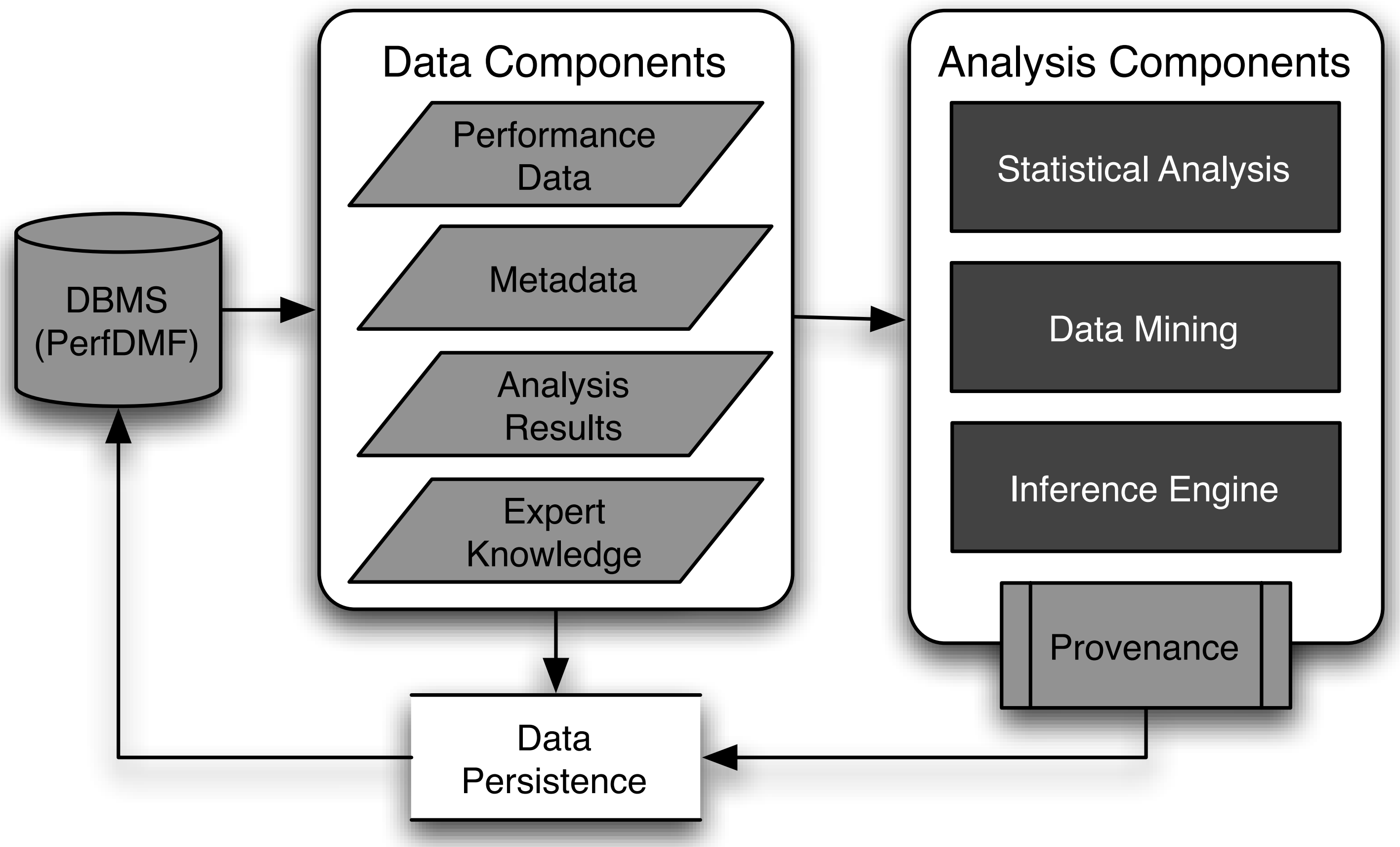

Figure 2: PerfExplorer components and their interactions.

```
# create a rulebase for processing
ruleHarness = RuleHarness.useGlobalRules("rules/GeneralRules.drl")
# load a trial
trial = TrialMeanResult(Utilities.getTrial("gtc", "jaguar", "512"))
# calculate the derived metric
fpOps = "PAPI_FP_INS"
time = "P_WALL_CLOCK_TIME"
oper = DeriveMetricOperation.DIVIDE
operator = DeriveMetricOperation(trial, fpOps, time, oper)
derived = operator.processData().get(0)
# compare values to average for application
mainEvent = trial.getMainEvent()
for event in derived.getEvents():
    MeanEventFact.compareEventToMain(derived, mainEvent, derived, event)
# process the rules
ruleHarness.processRules()
```

Figure 3: Sample Jython script.

```
rule "Poor L2 Hit rate"
    when
        // there is a L2 Cache hit rate lower than the average L2 Cache Hit rate
        f : MeanEventFact (
            m : metric == "((PAPI_L1_TCM-PAPI_L2_TCM)/PAPI_L1_TCM)",
            b : betterWorse == MeanEventFact.WORSE,
            s : severity > 0.10,
            e : eventName,
            a : mainValue,
            v : eventValue  )
    then
        System.out.println("The event " + e + " has a lower than average L2 hit rate.");
        System.out.println("\tAverage L2 hit rate: " + a + ", Event L2 hit rate: " + v);
        System.out.println("\tPercentage of total runtime: " + f.getPercentage(s));
end
```

Figure 4: Sample JBoss Rules rule.

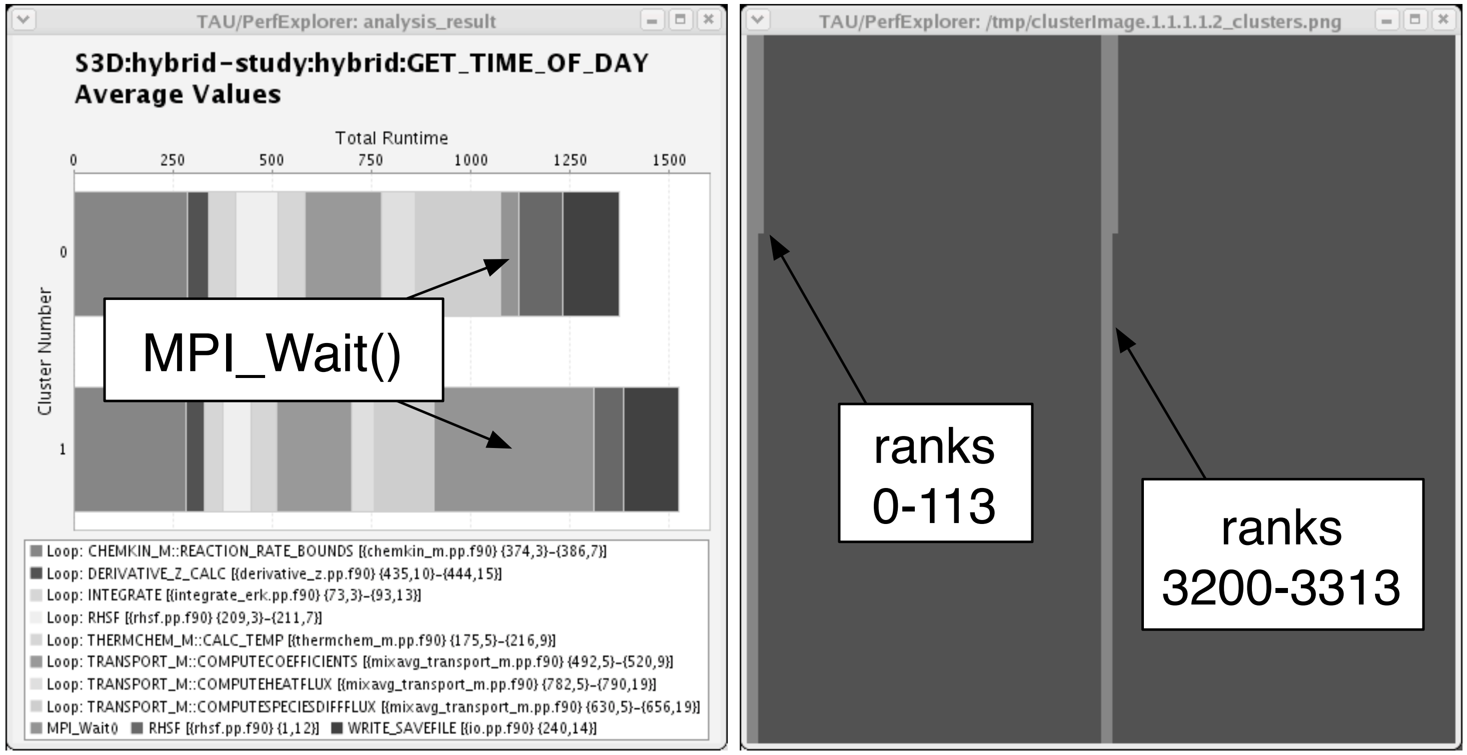

Figure 5: S3D cluster analysis. The figure on the left shows the difference in (averaged mean) execution behavior between the two clusters of processes. The figure on the right shows a virtual topology of the MPI processes, showing the locations of the clustered processes. the red processes ran on XT3 nodes, and the blue processes ran on XT4 nodes.

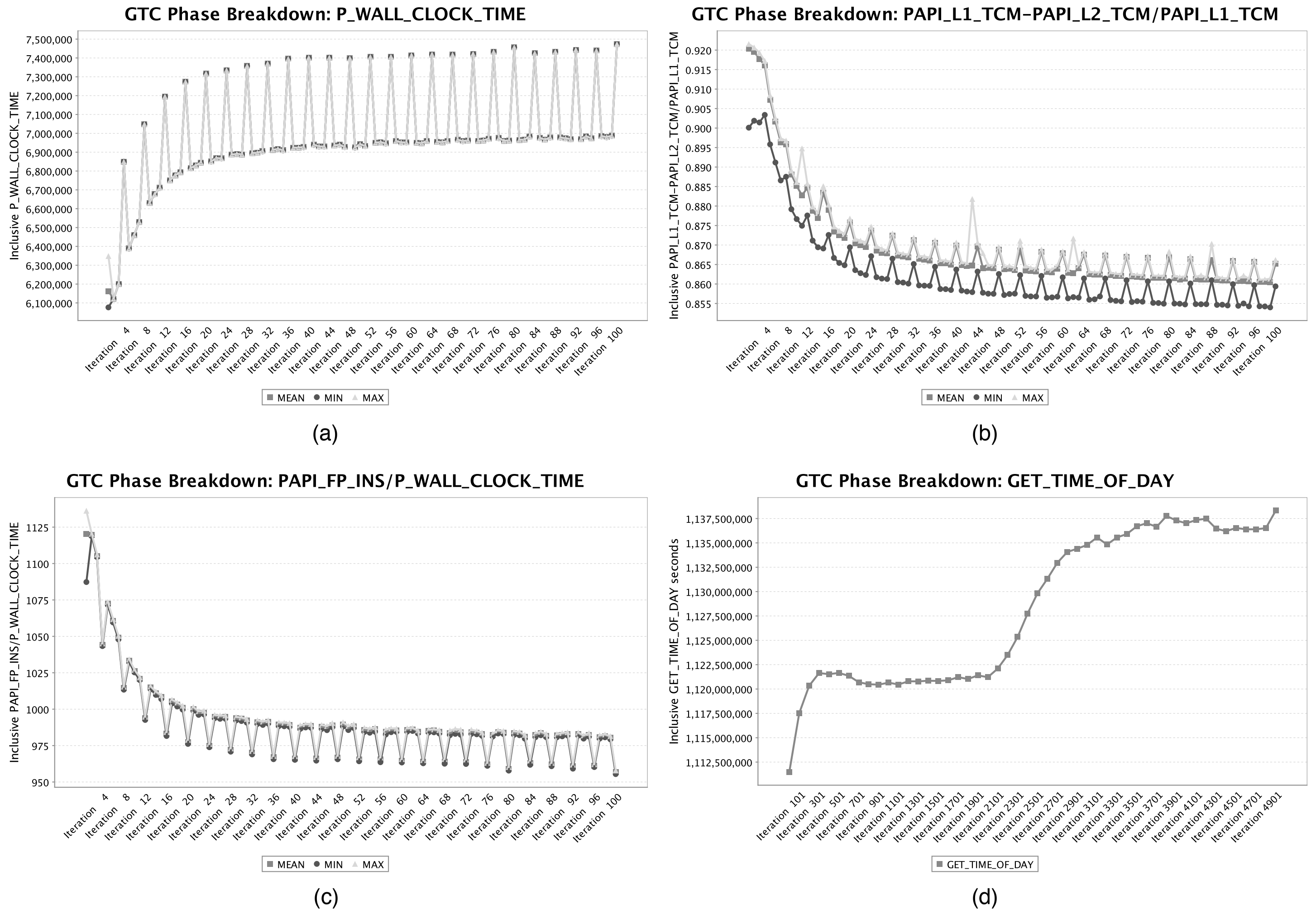

Figure 6: GTC phase analysis. (a) shows the increase in runtime for each successive iteration, over 100 iterations. (b) shows the decrease in L2 hit ratio, from 0.92 to 0.86, and (c) shows the decrease in GFLOPs from 1.120 to 0.979. (d) shows the larger trend when GTC is run for 5000 iterations, each data point representing an aggregation of 100 iterations.