An Algebra for Cross-Experiment Performance Analysis*

Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore

University of Tennessee, ICL
1122 Volunteer Blvd Suite 413
Knoxville, TN 37996-3450, USA
{song, fwolf, bhatia, dongarra, shirley}@cs.utk.edu

Abstract

Performance tuning of parallel applications usually involves multiple experiments to compare the effects of different optimization strategies. This article describes an algebra that can be used to compare, integrate, and summarize performance data from multiple sources. The algebra consists of a data model to represent the data in a platform-independent fashion plus arithmetic operations to merge, subtract, and average the data from different experiments. A distinctive feature of this approach is its closure property, which allows processing and viewing all instances of the data model in the same way - regardless of whether they represent original or derived data - in addition to an arbitrary and easy composition of operations.

Keywords: performance tool, multiexperiment analysis, tool interoperability, performance algebra, visualization

1 Introduction

Performance optimization of parallel applications usually involves multiple experiments to compare the effects of different code versions, different execution configurations, or different input data. For example, the experiments may reflect different algorithms, different domain decompositions, or different problem sizes. In addition, hardware characteristics may limit the availability of certain performance data, such as performance counters, in a single run, requiring multiple experiments to obtain a full set of data. Other than that, a user may wish to combine the results obtained using different monitoring tools that cannot be applied simultaneously. Also, the influence of system noise often creates a need to run the same experiment more than once in order to smooth the effect of random errors. Finally, data coming from analytical models or simulations constitute another class of data that need to be compared to those already mentioned.

The traditional practice of comparing different experiments is to put multiple single-experiment views side by side or to plot overlay diagrams, which is limited in its ability to draw a differentiated picture of performance changes because the hierarchical structures of the performance space, as they exist between performance metrics or program and system entities are typically ignored. A comprehensive and generic approach to extract important cross-experiment information along resource hierarchies is the framework for multi-execution performance tuning by Karavanic and Miller [11], which includes an operator to calculate a list of resources showing a significant discrepancy between different experiments. However, this difference operator maps from its input space containing entire experiments into a smaller representation (i.e., a list of resources). A repeated application is not possible, further processing would require a logic or a display different from one suitable for the original input data.

The key idea of our approach is that the output of cross-experiment analysis can be represented just like its input, which allows us to use the same set of tools to process and, in particular, display it. This article describes an algebra named CUBE (CUBE Uniform Behavioral Encoding) that can be used to compare, integrate, and summarize performance data of message-passing and/or multithreaded applications from multiple experiments including results obtained from simulations and analytical modeling. CUBE instantiates and extends the aforementioned framework by Karavanic and Miller and consists of a data model to represent the data in a platform-independent fashion plus arithmetic operations to subtract, merge, or average the data from multiple experiments. Our main contribution is that all operations are closed in that their results are mapped into the same space, yielding an entire "derived" experiment including data and

*This work was supported 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

metadata. As an important consequence of the closure property, we have been able to build an interactive viewer that allows convenient browsing through all elements of this space - regardless of whether they represent original experiments or derived experiments obtained by applying operations to original data. In addition, we can easily define composite operations, for example, in order to compute the difference of averaged data.

The article is organized as follows: In Section 2, we discuss the CUBE data model, followed by a description of the algebraic operations that can be performed on its instances in Section 3. After introducing the CUBE display component in Section 4, we demonstrate our method's usefulness using two practical examples in Section 5. Finally, we consider related work in Section 6 and present our conclusion plus future work in Section 7.

2 Data Model

Most performance data sets representing experiments, models, or simulations have a similar structure in that they are essentially mappings of program and system entities, such as functions and processes, onto a domain defined by one or more performance metrics, such as time and floating-point operations. The purpose of the CUBE data model was to give those commonalities a formal frame suitable for the definition of complex arithmetic operations.

The CUBE *data model* describes entity types to which performance data can refer, relationships between entities, and constraints that must be satisfied by a valid model instance. It consists of three different dimensions: a metric dimension, a program dimension, and a system dimension. Motivated by the need to represent performance behavior on different levels of granularity as well as to express natural hierarchical relationships among program and system resources, each dimension is organized in a hierarchy. A severity function determines how the different entities are mapped onto the actual metric values.

The metric dimension is represented as a forest consisting of multiple trees. Each metric has a name and a unit of measurement, which can either be seconds, bytes, or number of event occurrences. Within each tree, all metrics must have the same unit of measurement. To qualify for parentship of another metric, a metric must include the child metric. For example, execution time includes communication time and cache accesses include cache misses. Making a tool aware of this inclusion relationship by arranging metrics in a tree has the advantage that exclusive metrics can be computed automatically, for example, cache hits can be computed by subtracting misses from accesses.

The program dimension describes the static and dynamic program structure. Entities that can be defined include modules, regions, call sites, and call-tree nodes (i.e., call paths). A region is a general code section representing a function, a loop, or another type of basic block. Regions must be properly nested. Although a call site is called as such, it is more general and denotes a source-code location where the control flow may move from one region into another region. For example, a loop entry point is a call site according to our definition. The region that can be reached by executing the call site is called its callee. The set of all call-tree nodes may form a forest with multiple root nodes, but in most cases there will be a single root node representing the invocation of the main function. A parallel program with different executables, however, may need more than one root. Every call-tree node points to the call site from where the node was entered. Note that there may be multiple nodes pointing to the same call site. Recursive programs, whose call structure is a possibly cyclic graph and not a tree, must be mapped onto a tree, for example, by collapsing loops in the graph into a single leaf node.

The system dimension defines hard- and software entities of the system on which the program is executed. It is a forest consisting of the levels machine, node, process, and thread from top to bottom. A machine is a collection of nodes and can be a cluster or a massively parallel processor, such as the CRAY T3E. A node may host multiple processes, which can be split up into multiple threads. To simplify the merging of system hierarchies, the model more or less disregards the physical characteristics of machines and nodes and considers them mainly as a logical grouping of processes for the purpose of aggregating performance data. Nested thread-level parallelism is currently not supported. Since the thread level is mandatory, pure message-passing applications are represented as a collection of single-threaded processes.

Experiments. A valid instance of this data model is called a CUBE *experiment* and consists of two parts: metadata and data. The metadata part defines a set of metrics plus sets of program and system resources, as prescribed by the data model. As the data model allows a hierarchical arrangement of the different dimensions, ordering relations between their elements are included as well. The data part includes a *severity* function that maps tuples (metric m, call path c, thread t) onto the accumulated value of the metric m measured while the thread t was executing in call path c.

The severity function requires that data are mapped onto the call tree. Many performance tools, however, generate data referring to regions instead of call paths, that is, they generate flat profiles. This is not really a restriction, since every flat profile can be represented using multiple trivial call trees (one for each region) consisting only of a single node. Also, the severity of a certain tuple may be negative if it represents a difference between two experiments.

Data Format. CUBE experiments can be stored in the CUBE XML format. The format is specified using the XMLSchema language. A file representing a CUBE experiment consists of two parts: the metadata and the severity function values. The severity values are stored as a three-dimensional array with one dimension for the metric, one for the call path, and one for the thread. We have implemented a C++ API to read experiments from a file and to create experiments and write them to a file. The API is a simple class interface with fewer than fifteen methods.

3 Operations

In addition to reading and writing experiments, we allow experiments to be transformed by applying certain operators. The domain of all operators is the set of valid CUBE experiments. The range is always a subset of the domain so that the domain is never left. That is, the output of an operator can always be used as input for another operator, enabling the simple specification of complex operations by creating composite operators. To distinguish original data that has been collected during a real experiment from data that is the result of an operator we call the latter experiment a *derived* experiment as opposed to an *original* experiment.

Operators. We have defined three algebraic operators that we deem most useful for performance analysis. Others may follow in the future.

- Difference
- Merge
- Mean

The mean operator takes an arbitrary number of arguments, whereas the difference and merge operator are binary operators. However, since their range is a subset of the domain, a user can construct composite operators involving far more than two operands.

The difference operator takes two experiments and computes a derived experiment whose severity function reflects the difference between the minuend's severity and the subtrahend's severity. This feature is useful to compare the effects of code or parameter changes along the different dimensions of the data model.

The merge operator's purpose is the integration of performance data from different sources. Often a certain combination of performance metrics cannot be measured during a single run. For example, certain combinations of hardware events cannot be counted simultaneously due to hardware resource limits. Or the combination of performance metrics requires using different monitoring tools that cannot be deployed during the same run. The merge operator takes two CUBE experiments with a different or overlapping set of metrics and yields a derived CUBE experiment with a joint set of metrics.

On parallel systems, unrelated system activities often perturb performance experiments in a way that lets results vary across multiple executions. For example, the execution time of a program can be different for separate runs even if all user-controlled execution parameters remain stable. Also, a user might want to combine several execution parameters in an overall picture in order to make a single statement about the performance for a range of execution parameters. The mean operator is intended to smooth the effects of random errors introduced by unrelated system activity during an experiment or to summarize across a range of execution parameters. To summarize in this manner, the user can conduct several experiments and create one derived experiment from the whole series.

Implementation. The actions performed by the operators can be divided into two subtasks: metadata integration followed by the actual arithmetic operation. It is obvious that if the metadata of the two operands are equal (i.e., if the structure of the metric, the program, and the system dimension is the same) the operation is reduced to a simple arithmetic operation on corresponding elements of the three-dimensional severity arrays. In the case of the difference operator this corresponds to an element-wise subtraction and in the case of the mean operator to an element-wise mean operation. As the merge operator is intended to join experiments involving different metrics, there is no simple case for this one.

However, the general case we have to deal with is two experiments with different metadata. In most cases the operators make sense only if there is at least some similarity between them. For example, computing the mean of experiments that test entirely different programs is generally not helpful. On the contrary, the difference between program runs with slightly different call trees can help in comparing the performance of alternative program versions. The next subsection deals with the task of integrating two different metadata sets.

Metadata Integration. Before executing the actual arithmetic operation when applying an operator to multiple experiments, we first need to integrate their metadata sets. The integration of one or more metadata sets consists of three separate parts: merging the metric dimension, the program dimension, and the system dimension. Merging metric trees and call trees can be more or less reduced to the task of merging arbitrary trees. Except for a different equality relation to compare the nodes in a tree, the procedure is very similar. To do this, we use the multi-execution framework's [11] structural merge operator. While traversing from the

roots to the leaves, we try to match up the nodes from the two metadata sets using the equality relation. The equality relation is based on node attributes, such as name and unit of measurement for a metric or the callee for a node in the call tree. Nodes that cannot be matched are separately included in the new metadata set, whereas nodes that can be successfully matched become shared nodes, that is, they appear as a single node in the new metadata. The matching occurs in a top-down fashion with the consequence that once two nodes are considered different, the entire subtrees rooted at these nodes will both become part of the new metadata set even if they contain matching child nodes. When matching nodes in a call tree we have to take into account that certain call site attributes, such as line numbers, can change across different code versions but still refer to the "same" call site, a problem we have not addressed yet and which requires further consideration.

Integrating the system dimension is slightly different. Here, we have four levels with different meanings: machine, node, process, and thread. First, processes and threads are matched based on their application-level identifiers, for example, their global MPI rank and OpenMP thread number. The upper levels of the system hierarchy are not matched. Instead, CUBE either copies the entire node and machine hierarchy including the corresponding process-node mapping of one of the operand experiments into the result experiment or it collapses the hierarchy to a single machine and a single node. If not specified otherwise, the latter option is the default if the partitioning of processes into nodes is not compatible among the operands.

The focus of CUBE is to provide automatic merging mechanisms that follow simple rules and create predictable results without requiring manual intervention. As the default behavior might not satisfy the user in all possible situation, switches have been included to change the default according to a user's needs.

Arithmetic Operation. After the metadata have been integrated, a new severity function is computed whose domain is defined by the integrated metadata. This happens by an element-wise operation on the two input arrays. To be able to perform an element-wise operation, the operand's severity function needs to be extended with respect to the integrated metadata so that it is defined for every tuple (metric, call path, thread) of the new metadata. This is done by assigning zero to previously undefined tuples. For example, a call path occurring in one metadata set might not occur in another. If this happens the resulting value for this call path will be set to zero in those experiments that didn't contain the call path before.

In the case of the difference or the mean operators, the element-wise operation is just a subtraction or mean operation, respectively. In the case of the merge operator we make a simple case distinction. Recall that the purpose of the merge operator was to integrate performance experiments with different metrics. For example, one experiment counts floating point operations, and another one counts cache misses, since we might not be able to count both of them simultaneously. So, if the metric is provided only by one experiment we take the data from that experiment. If it is provided by both experiments we take it from the first one without loss of generality.

Note that all operators return a complete (albeit derived) CUBE experiment consisting of an integrated metadata set and a severity function with the integrated metadata as its domain.

4 Display Component

A natural application of the CUBE performance algebra is visual presentation. The closure property allows us to treat derived experiments just like original ones. For this purpose, we have implemented the CUBE display, a generic viewer that provides the ability to interactively browse through the multidimensional performance space defined by any valid CUBE experiment, whether it is derived or original.

As acceptance of performance tools among program developers is often limited by their complexity [14], our design emphasizes simplicity by combining a small number of orthogonal features with a limited set of user actions. Similar to Paradyn, CUBE displays the different dimensions of the performance space consistently using tree browsers (Figure 1). More than that, CUBE allows the user to interactively explore the severity mapping of metric/resource combinations onto the corresponding values. Since the space of all such combinations is large, CUBE provides the ability to select a view representing only a subset of the mapping plus flexible aggregation mechanisms to control the level of detail. In addition, the GUI includes a source-code display that shows the exact position of a performance problem in the source code. The CUBE display is implemented in C++ using the wxWidgets GUI toolkit and libxml2 to parse the CUBE XML format. Currently, CUBE supports most UNIX platforms and a Windows version is in preparation.

Basic Principles. The CUBE display consists of three tree browsers, representing the metric, the program, and the system dimension from left to right (Figure 1). The user can switch between a call tree or a flat-profile view of the program dimension. The call-tree view, as shown in the figure, is the default. The nodes in the metric tree represent performance metrics, the nodes in the call tree represent call paths, and the nodes in the system tree represent machines, nodes, processes, and threads from top to bottom. The thread level of single-threaded applications is hidden.

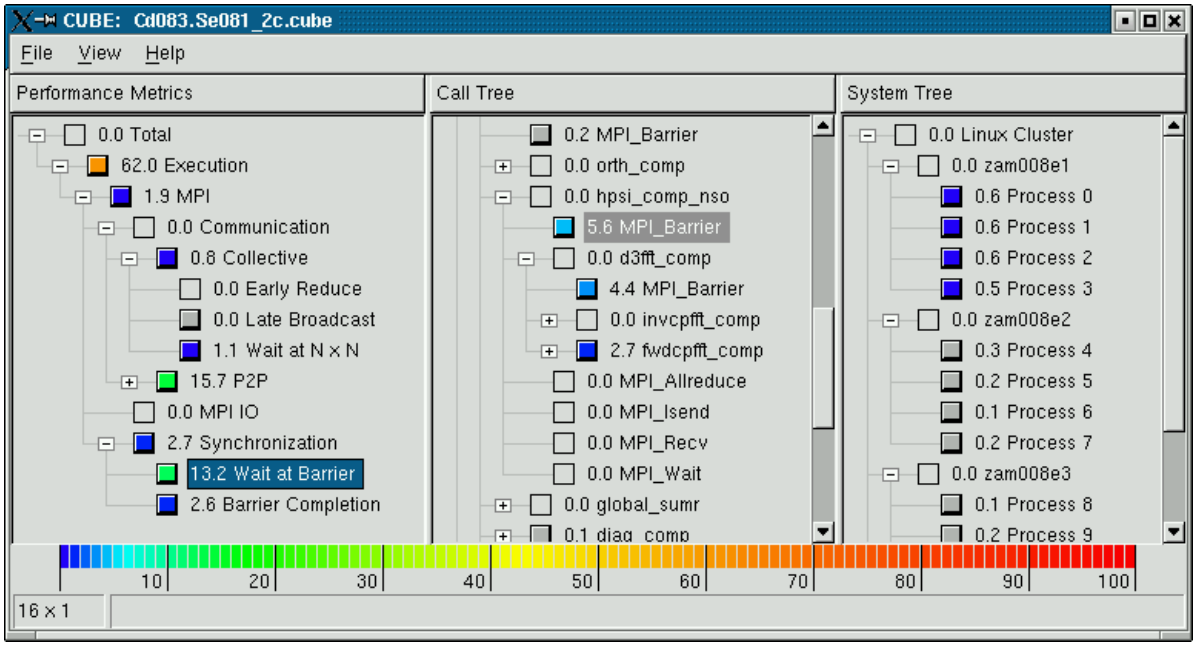

Figure 1. The CUBE display showing the metric tree, the call tree, and the system tree from left to right. The selected node in the metric tree indicates large waiting times in front of MPI barriers.

A user can perform two types of actions: selecting a node or expanding/collapsing a node. At any time, there are two nodes selected, one in the metric tree and another one in the call tree. Currently, it is not possible to select a node in the system tree. Each node is labeled with a severity value (i.e., a metric value). A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and the entire system. A value shown in the call tree represents the sum of the selected metric across the entire system for a particular call path. A value shown in the system tree represents the selected metric for the selected call path and a particular system entity. All numbers can be displayed either as absolute values or as percentages of a maximum value. Percentages can be normalized with respect to other experiments to simplify the comparison. To help identify metric/resources combinations with a high severity more quickly, all values are ranked using colors. While the color indicates the severity's absolute value, its sign is indicated by giving the square a relief: a raised relief for positive values and a sunken relief for negative values. Depending on the severity representation, a color legend shows a numeric scale mapping colors onto values.

Note that all hierarchies in CUBE are inclusion hierarchies, meaning that a child node represents a subset of the parent node. For instance, in Figure 1 Wait-at-Barrier, which denotes the waiting time in front of barriers, is a subset of Synchronization. The severity displayed in CUBE follows the principle of *single representation*, that is, within a tree each fraction of the severity is displayed only once. The purpose of this display strategy is to have a particular performance problem to appear only once in the tree and, thus, help identify it more quickly. In Figure 1, the value at Synchronization refers only to those synchronization times that are not covered by descendant nodes, that is, those that are neither waiting times in front or completion times after the barrier. After collapsing the Synchronization node, the label would show the entire time spent in MPI barriers. Thus, the display provides two aggregation mechanisms: aggregation across dimensions by selecting a node, and aggregation within a dimension by collapsing a node.

The emphasis of CUBE was not the invention of a new display in a technical sense. After all, the use of tree browsers is not revolutionary and even the coloring of nodes in the tree to symbolize a numeric value has been previously applied, for example, in the xlcb [15] corefile browser. However, CUBE demonstrates that the simplicity of the data model can be transferred into an interactive display with flexible view-selection capabilities by restricting the design to a very small number of orthogonal mechanisms.

5 Examples

This section demonstrates the advantages of our approach using two realistic examples. We illustrate how to conduct a before-after comparison by browsing through the

difference of two experiments in the same way a user would browse through original data. We also show how CUBE can create a very comprehensive picture of performance behavior by combining data sets with different performance metrics recorded by different tools and using different modes of the same tool into one highly-integrated view.

CUBE is currently used by two performance tools: CONE [21] and EXPERT [22, 23]. As CUBE provides a generic API, every tool producing performance data matching the very general CUBE data model can take advantage of the CUBE algebra and display.

CONE. CONE is a call-graph profiler for MPI applications on IBM AIX Power4 platforms which maps hardware-counter data onto the full call graph including line numbers. CONE is based on a run-time call-graph tracking technique [7] developed at IBM. CONE automatically traverses and instruments the executable in binary form using DPCL [6] and causes the target application to make calls to a probe module responsible for performance monitoring. The performance data collected include wall-clock time as well as different hardware counters accessible via the PAPI library [4]. CONE supports several event sets that can be selected for measurement. Each of them forms a hierarchy of more general and more specific events, such as cache accesses and cache misses or instructions and floating-point instructions, respectively.

EXPERT. EXPERT is a post-mortem performance analysis tool for automatic analysis of MPI and/or OpenMP traces. Time-stamped events, such as entering a function or sending a message, are recorded as the target application runs and are later written to a trace file in the EPILOG format. After program termination, the trace is searched for execution patterns that indicate inefficient behavior. The performance problems addressed include inefficient use of the parallel programming model and low CPU and memory performance. EXPERT transforms event traces into a compact representation of performance behavior, which is essentially a mapping of tuples (performance problem, call path, location) onto the time spent on a particular performance problem while the program was executing in a particular call path at a particular location. Depending on the programming model, a location can be either a process or a thread. The performance problems are organized in a specialization hierarchy that contains general problems, such as large communication overhead, and very specific problems, such as a receiver waiting for a message as a result of an inefficient acceptance order. After the analysis has been finished, the mapping is stored in the CUBE format.

5.1 Subtracting Performance Data

Changing a program can alter its performance behavior. Altering the performance behavior means that different results are achieved for different metrics. Some might increase while others might decrease. Some might rise in certain parts of the program only, while they drop off in other parts. Finding the reason for a gain or loss in overall performance often requires considering the performance change as a multidimensional structure. With CUBE's difference operator, a user can view this structure by computing the difference between two experiments and rendering the derived result experiment like an original one.

As an example, we show how the CUBE difference operator can help track the effects of an optimization applied to PESCAN [5], a nano-structure simulation computing interior eigenvalues nearest to a given point of a (large) Hermitian matrix. The core of the application consists of an iterative eigensolver based on the preconditioned conjugate gradient method applied to the folded spectrum. The type of computation performed is a matrix-vector products done via FFT.

We conducted the experiments on a Intel Pentium III Xeon 550 MHz cluster with eight 4-way SMP nodes connected through Myrinet. We ran the application with 16 processes on four of the nodes to compute a medium-sized particle model. Figure 1 shows the CUBE display with a data set obtained from the unoptimized code version. The numbers reflect percentages of the overall execution time. The selected metric in the left tree represents a major performance problem: A large fraction of the execution time is spent waiting in front of barriers (13.2 %).

Wait-at-Barrier denotes the time a process waits inside the barrier function for another process to reach it as opposed to the time spent in the barrier function after the first process has left it (i.e., Barrier-Completion) or to collectively execute it (i.e., everything in between). Note that the distinction between these aspects of barrier synchronization requires measuring temporal displacements within individual barrier instances as they are recorded in trace files. Since Wait-at-Barrier is selected, the call tree shows the locations of barriers with excessive waiting times highlighted by colors.

The barriers have originally been introduced to avoid buffer overflow related to the asynchronous point-to-point communication when computing with large processor counts on an IBM platform. However, since they are not needed on a Linux cluster with smaller processor counts, such as those used in this experiment, we were able to speed up the application by removing them.

Waiting time at a barrier is caused by reaching the barrier at different points in time, for example as a result of load or communication imbalance or other delays. Some of the factors introducing temporal displacements are antipo-

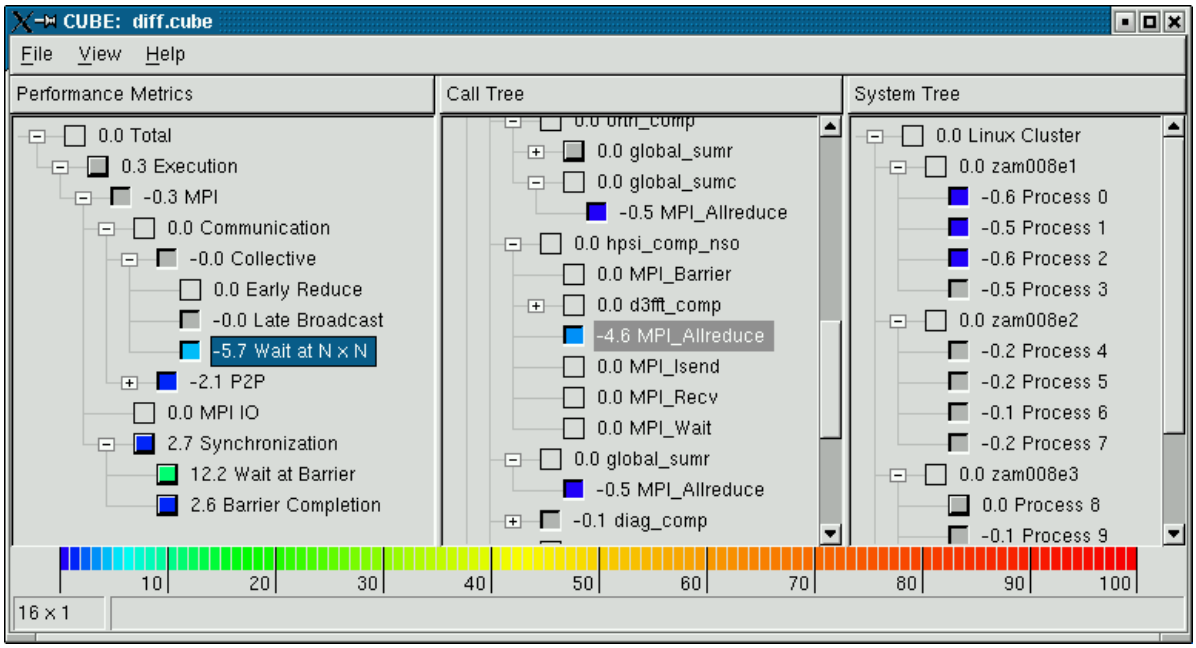

Figure 2. A difference experiment shows the disappearance and migration of waiting times for application PESCAN.

dal and cancel each other out if they are not materialized at a barrier or another synchronizing event. If they are materialized, the time that a process reaches the barrier earlier is turned into waiting time, which is effectively lost because after the synchronizing event the processes have caught up with each other. Worse than that, the contrary displacement might be materialized as well instead of neutralizing the previous one. So removing the barriers can save the actual barrier overhead needed to perform the synchronization as well as allow contrary displacements to cancel each other out. Those delays or imbalances that do not have an antipodal counterpart are usually materialized at the next synchronization point following the removed barrier, which can be a message exchange or a collective operation.

Figure 2 shows a difference experiment obtained by subtracting the optimized version from the original one. Performance gains are represented by raised reliefs, performance losses by sunken reliefs. The numbers are normalized with respect to the old version and show improvements in percent of the previous execution time. Whereas nearly all the negative performance effects of barrier synchronization have been eliminated including waiting time, barrier execution, and barrier completion, point-to-point communication (i.e., P2P) and inherent synchronization of collective all-to-all operations (i.e., Wait-at-NxN) have been increased, presumably as a result of waiting-time migration.

However, the gross performance balance is clearly positive. We measured the performance gain for the central solver routine only without any trace instrumentation. We created two series of ten experiments for either configuration and took the minimum of each series as a representative. The speedup obtained for the solver by removing the barriers was about 16 %.

The waiting time still present in the application reflects to some extent a computational load imbalance as can be seen when viewing how execution time without MPI calls is distributed across the different processes (not shown here).

5.2 Integrating Performance Data

An integration of trace-based analysis to target parallel performance with counter-based analysis to target memory performance has proved to be a useful strategy [24] in order to define bounds for the runtime penalty of cache misses. An above average cache miss rate was found in MPI calls of the SWEEP3D benchmark code [2] using EXPERT. At the same time, those MPI call were identified as the source of Late-Sender problems with the result that most of the time spent in those calls was waiting time anyway, rendering the cache-miss problem insignificant. The monitoring method applied was recording the number of cache misses as part of individual trace records. However, recording one or more hardware-counter values as part of nearly every event record can increase trace-file size dramatically, a side-effect that severely limits the scalability of this approach. In this particular case, the timestamped storage of hardware-counter

data was even unnecessary, since EXPERT did nothing other than accumulating the counters for every call path.

Taking advantage of the CUBE merge operator, it is now possible to record hardware-counter and trace data separately. In this way, counter data can be collected as a less space-intensive call-graph profile from the very beginning using CONE, thereby avoiding the undesired trace file enlargement in addition to dividing the overall measurement overhead between two program runs. Since both CONE and EXPERT produce CUBE-compliant output, we can use the merge operator to obtain a single derived CUBE experiment integrating the output from two different sources.

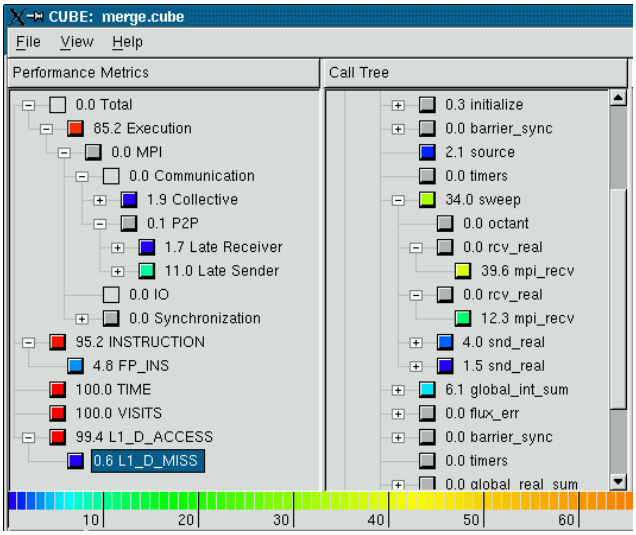

Figure 3. Merge of outputs from CONE and EXPERT.

If we want to consider multiple counters in our analysis, we might have to deal with a hardware restriction limiting the number and type of counters to be monitored in parallel. POWER4, for example, does not permit the combination of floating-point instructions with level 1 data-cache misses in the same run. The solution above can, of course, be applied to different outputs from the same tool as well. In this case, we can perform different measurements with CONE using different event sets and merge the results into a single experiment. To alleviate the effects of random errors, we can summarize multiple outputs from every single tool by applying the mean operator before we perform the merge operation.

Figure 3 shows a derived experiment obtained by merging one EXPERT output with two CONE outputs referring to different event sets. It includes metrics from two tools for the same application. The first tree from the top is the root of EXPERT's metric hierarchy. It is expanded and shows different trace-based metrics. Below are the counter-based metrics from CONE including level 1 data cache misses (L1_D_MISS) and floating-point instructions (FP_INS), which have been collected during two different runs. The numbers represent percentages of their corresponding root metric's total amount. The call tree shows the percentage distribution of cache misses with a high concentration at MPI_Recv calls which are at the same time sources of Late-Sender problems (not shown).

6 Related Work

This work builds upon the framework for multi-execution performance tuning by Karavanic et al. [11], which was used in the Paradyn project [13] for an optimization strategy based on using historical performance data to guide the search for performance bottlenecks. Similar to CUBE, the framework defines operations on performance data stored in accordance with a specific data model. Whereas the framework's data model defines experimental data in terms of arbitrary resource hierarchies, CUBE favors interoperability over flexibility by adding more semantics and exactly specifying the type of program and system resources that can be described. Also, the framework does not consider any relationships between performance metrics, whereas CUBE models specialization relationships between metrics in the style of dependence relationships between the search hypotheses in Paradyn. The most significant difference between CUBE and the framework is that all operations in CUBE are closed in that an operator always maps its operands back into its domain. The framework includes a structural merge and a structural difference operator which, however, is defined only for experiment metadata as opposed to the actual performance numbers. In contrast, the performance difference operator, which computes the discrepancy between the actual performance data of two experiments, returns a list of foci (i.e., combination of resources from different hierarchies) where this discrepancy is significant. CUBE's difference operator and all other operators are defined for entire experiments and return entire (albeit derived) experiments including actual performance numbers that can be processed and displayed like original ones. For example, mechanisms aimed at finding hotspots can be applied to the original and the difference data likewise. Also, the framework does not include a mean operator. Another difference is that CUBE offers an API to transfer data from arbitrary sources into its specific input representation.

The problem of multiexperiment analysis has been addressed by many other groups as well. Projects such as ILAB [25], NIMROD [1], Tuner's Workbench [10], and ZENTURIO [17] provide an infrastructure for planning and conducting a series of experiments with different parameters. ZENTURIO can be combined with the AKSUM [19] perfor-

mance tool to automatically analyze the evolution of higher-level performance problems across multiple experiments. Scalability-analysis features can also be found in the latest release of SvPablo [18].

There are a multitude of interactive browsers, such as AKSUM [19], HPCView [12], and SvPablo [18], that correlate the program structure with different performance metrics. The CUBE display's distinctive feature is the combination of a generic API that makes it available for third-party tools with a set of operations that can be performed on the data. In addition, CUBE's presentation logic achieves simplicity by relying on a single type of widget (i.e, a tree browser) regardless of the performance-space dimension displayed. Historically, CUBE emerged from the KOJAK [23] project, where a similar browser was used to present the results of event-trace analysis.

The CUBE algebra is based on a generic model of performance data providing a foundation for many postprocessing tools. The ASL [9] specification language, which provides language constructs to specify potential performance problems in parallel applications, is based on a similar data model with emphasis on automatic problem detection.

As CUBE focuses on automatic postprocessing of a wide range of performance data, it is also closely related to performance database projects, such as PerfDBF [8] and PPerfDB [16]. In fact, implementing the CUBE algebra on top of a database management system in addition to a pure XML file representation would be a natural extension, and interfacing to an existing performance database might open a large amount of performance data to our approach. On the other hand, CUBE - by relying on XML files only - provides cross-experiment capabilities without the burden of maintaining a whole database-management system.

7 Conclusion

The CUBE performance algebra addresses the problem of analyzing multiple performance data sets by defining a data model to represent a wide range of performance experiments plus operations to compare, integrate, and summarize them. The closure property of the algebra enables a new level of tool interoperability by combining the views provided by different experiments and performance tools into a single one and making this integrated view available to visualization and other postprocessing tools.

We have implemented a library to read and write experiments and to perform arithmetic operations on them, which is currently used by CONE and EXPERT. A generic display component illustrates the enriched view provided by derived experiments. Browsing through a difference experiment allows us to analyze the effects of optimizations in a very differentiated manner. Also, merging tracing output with profiling output can help reduce trace-file size significantly. Finally, hardware restrictions limiting the number of hardware counters measured simultaneously can now be circumvented by merging the outputs of separate experiments.

We believe that our approach is especially well suited to support performance analysis on large-scale systems. As parallel machines grow larger, monitoring becomes much more complicated. Given the enormous amount of data generated even for a single metric during a single run, the ability to automatically integrate data from different runs becomes more and more important. New operators which perform data reduction, for example, based on multivariate statistical techniques [3], might further help manage size when applied to the integrated data. Also, the integration of topology information, for example obtained from instrumented MPI topology routines, into our data model could open the way for new automatic analysis and visualization tools. Finally, as the processing logic of CUBE relies entirely on an XML-centric data model, CUBE can be easily integrated with an Grid environment by exposing its functionality as an OGSI-compliant Grid service [20].

Acknowledgment. We would like to thank Andrew Canning and Lin-Wang Wang from Lawrence Berkeley National Lab for giving us access to their application. We are also grateful to Julien Langou for helping us run the PESCAN code and explaining to us the mathematics behind it.

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parameterised Simulations Using Distributed Workstations. In *Proc.of the 4th IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 112–121, 1995.

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

[3] D. H. Ahn and J. S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. In *Proc. of the Conference on Supercomputers (SC2002)*, Baltimore, November 2002.

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

[5] A. Canning, L.W. Wang, A. Williamson, and A. Zunger. Parallel empirical pseudopotential electronic structure calculations for million atom systems. *Journal of Computational Physics*, 160(29), 2000.

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

[7] L. A. DeRose and 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*, number 2400 in LNCS, pages 167–176, Paderborn, Germany, August 2002. Springer.

[8] J. Dongarra, A. Malony, S. Moore, P. Mucci, and S. Shende. Scalable Performance Analysis Infrastructure for Terascale Systems. *Future Generation Computer Systems*, 2004. To appear.

[9] T. Fahringer, M. Gerndt, B. Mohr, G. Riley, J. L. Träff, and F. Wolf. Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrum Jülich, August 2001. Revised version.

[10] A. Hondroudakis and R. Procter. The Tuner's Workbench: A Tool to Support Tuning in the Large. In P. Fritzson, editor, *Proc. of the ZEUS-95 Workshop on Parallel Programming and Computation*, pages 212–221. IOS Press, May 1995.

[11] K. L. Karavanic and B. Miller. A Framework for Multi-Execution Performance Tuning. *Parallel and Distributed Computing Practices*, 4(3), September 2001. Special Issue on Monitoring Systems and Tool Interoperability.

[12] 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.

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

[14] C. M. Pancake. Applying Human Factors to the Design of Performance Tools. In *Proc. of the 5th International Euro-Par Conference*, number 1685 in LNCS, pages 44–60. Springer, August/September 1999.

[15] Parallel Tools Consortium. *xlcb Graphical Browser: User Manual*. `http://web.engr.oregonstate.edu/~pancake/ptools/lcb/xlcb.html`.

[16] Portland State University. *PPerfDB*. `http://www.cs.pdx.edu/~karavan/research.html`.

[17] R. Prodan and T. Fahringer. On Using ZENTURIO for Performance and Parameter Studies on Cluster and Grid Architectures. In *Proc. of 11th Euromicro Conf. on Parallel Distributed and Network based Processing (PDP 2003)*, Genua, Italy, February 2003.

[18] L. A. De Rose, Y. Zhang, and 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, September 1999.

[19] C. Seragiotto Júnior, M. Geissler, G. Madsen, and 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, February 2003.

[20] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. *Open Grid Services Infrastructure (OGSI) Version 1.0*. Global Grid Forum, June 2003. Draft Recommendation.

[21] University of Tennessee. *CONE*. `http://icl.cs.utk.edu/kojak/cone/`.

[22] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003. ISBN 3-00-010003-2.

[23] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".

[24] F. Wolf and B. Mohr. Hardware-Counter Based Automatic Performance Analysis of Parallel Programs. In *Proc. of the Minisymposium "Performance Analysis", Conference on Parallel Computing (ParCo)*, Dresden, Germany, September 2003.

[25] M. Yarrow, K. M. McCann, R. Biswas, and R. F. Van der Wijngaart. An Advanced User Interface Approach for Complex Parameter Study Process Specification on the Information Power Grid. In *Proc. of GRID*, pages 146–157, 2000.