

PerfMiner: Cluster-wide Collection, Storage and Presentation of Application Level Hardware Performance Data

Philip J. Mucci^{1,2}, Daniel Ahlin², Johan Danielsson², Per Ekman², and Lars Malinowski²

¹ Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA,

² Center for Parallel Computers, Royal Institute of Technology, Stockholm, Sweden

Abstract. We present PerfMiner, a system for the automated collection, storage and presentation of hardware performance metrics gathered per thread via PAPI, across an entire cluster. In PerfMiner, every sub-process/thread of an application is measured with essentially zero overhead. Furthermore, the collected metrics are precise per thread measurements that are unaffected by other kernel or user processes. PerfMiner correlates this performance data with metadata from the batch system in a scalable database to provide detailed knowledge about numerous aspects of the system’s performance. Through a command line and Web interface, queries to this database can quickly and easily answer questions that range from high level workload characterization to low level thread performance. Other monitoring systems merely provide an overall view of the system by reporting aggregate system-wide metrics sampled over a period of time. In this paper, we describe our implementation of PerfMiner as well as present sample results from the test deployment of PerfMiner across three different clusters at the Center for Parallel Computers at The Royal Institute of Technology in Stockholm, Sweden.

1 Introduction

The desire to understand the behavior of applications and their performance on large scale systems has always existed. As long as unlimited computing power is not freely available, large systems must be carefully managed in order to allow scientists and engineers to perform their computations within a given time frame. In most modern supercomputing installations, the cost of the machines, software and maintenance is passed along to these users in terms of allotted time. This time is either directly purchased by the user or applied for and granted by a central authority; usually the same agency that funds the purchase and operation of the machine. This process is designed to balance a budget, basically equating allocated CPU hours of computational science with the cost of installation and operation of a large machine. Given that the lifetime of modern processors is three to four years before being made obsolete via Moore’s Law, we find the above process to be wholly unsatisfactory as it makes no attempt to optimize the use of either financial or computational resources. Compute time from user

to user, group to group is treated equally, even though the amount of work that can be accomplished during each CPU hour can differ by many orders of magnitude. For example, a user with a large allocation and a tremendously inefficient code can easily 'steal' otherwise available resources from less well-funded research groups. The allocation is not based on actual computational 'work' nor efficiency, but rather a rough estimate on the number of hours that can be purchased with a given budget. Given the same budget, it is conceivable that this very same user could solve problems on a much larger scale with an optimized code. The converse does not necessarily hold, as a user with a small budget and a large problem must strive to achieve some degree of efficiency in order to complete his work in the allotted time. If the allocation policy was biased towards demonstrated computational demand or towards the efficient use of resources, aggregate throughput of the system would rise and more CPU hours would be available to the scientific community as a whole. The above dilemma represents only one small piece of the problem regarding the understanding the actual usage of these large systems. Consider these other cases:

Purchase of a New Computing Resource. Procurements are often run in two different modes; either the customer submits a set of benchmarks to be optimized and run by the vendor or the vendor provides access to hardware resources on which the customer runs the benchmarks. These benchmarks run the gamut from microbenchmarks that measure particular machine parameters to full-blown applications. However, benchmarks by their very nature, attempt to represent a very large code base with a very small one. If hardware performance data could be collected for every application and correlated with data from the batch system, specific criteria that bound application performance could be used to guide the procurement process. For example, answers to questions like "Do the majority of our applications demonstrate high level 2 cache hit rates and thus are sensitive to cache size and front side bus frequency?" provide specific information about what kind of hardware should be purchased.

Improving the Software Development Cycle. While there are many excellent open source performance analysis tools available[TAU][SvPablo][Paradyn][Mucci], virtually all of them require the user to change his environment, Makefiles or source code. While simple for computer scientists, this process is fraught with potential error for scientists and engineers who are focused on getting their research done. One or two failed attempts at using a tool is enough to permanently deter a scientist from making further efforts to characterize or understand his application. If the monitoring system could itself provide a completely transparent mechanism to measure important performance characteristics and the user could access that information quickly and easily, the process of application understanding could become second nature to the user and part of the software development process itself.

Performance Focused System Administration. As mentioned above, by having access to performance data about all applications, system administrators could

systematically address applications and their users that make inefficient use of compute resources. Centers with application specialist teams could deploy staff on the basis of low performance and high CPU hour consumption. This type of targeted optimization effort has the potential of optimizing a sites heavy users and reap continued benefits through successive generations of machines as the big users' applications receive the attention they deserve.

1.1 The Design of PerfMiner

In order to meet the above needs, a performance collection system must be carefully designed. First and foremost, it must be focused on the simplicity of it's user interface and the speed of which it operates. As the system could be running on many clusters across a site and measure every job through the system, the amount of data could grow quite large. The system has four basic components: Integration into the user's environment and/or batch system. This must be completely transparent to the user, but yet facilitate conditional execution of monitoring for debugging and other purposes. Collection of the job and hardware performance data. This must also be completely transparent to the user with no modifications to the user's job. Post-processing of the data and insertion into a database. The database must be carefully designed to support queries that may span tables with ten's of millions of rows. Furthermore, the schema should facilitate the rapid development of reasonably complex queries in order to accommodate the demands of its user base. Presentation of the data to the users, system-administrators and managers. This interface must be as simple as possible to guarantee maximum acceptance into a daily usage cycle. Complex functionality should be hidden from the main interface yet remain accessible to those wishing to dig deeper. The interface should facilitate rapid drill-down investigation from widest granularity down to the thread level.

PerfMiner is an performance monitoring system that attempts to meet the above goals. It is being actively developed at the Center for Parallel Computers (PDC) at the Royal Institute of Technology in Stockholm, Sweden. To test our initial implementation, we deployed PerfMiner for a subset of users for three weeks across all three of PDC's clusters, Roxette, a cluster of 16 dual Pentium III nodes, Lucidor, a cluster of 90 dual Itanium 2 systems, and Beppe, a 100 processor Pentium IV cluster that is one of the six SweGrid clusters spread across Sweden. All systems have gigabit ethernet as an interconnect, with the exception of Lucidor which also contains Myrinet-D cards in every node.

In the next four sections, we describe each of the components of the PerfMiner system in detail. Working our way from the integration into the batch system to the Web interface presented to the user. In Section 6, we present the results of a few queries and the resulting data's relevance to the various users of the PerfMiner system. We then discuss how PerfMiner relates to previous work in Section 6 and make some conclusions and discuss our future plans for PerfMiner in Section 7.

2 Integration of PerfMiner into the Easy Batch System

One of the challenges of the implementation of PerfMiner at PDC was how to manage the integration into the batch system. PDC runs a modified version of the Easy[Easy] scheduler. At its core, Easy is a reservation system that works by enabling the user's shell in `/etc/passwd` on the compute nodes. The user is free to login directly to any subset of the reserved nodes. There is no restriction on using MPI as a means to access these nodes from the front end. In this way, Easy serves the needs of PDC's data processing community who frequently submit ensembles of serial jobs, often written in Perl or Bourne shell. Given this, we could not count on mpirun as our single point of entry to the compute nodes. This left us with only one means to guarantee the initiation of the collection process: the installation of a shell wrapper as the user's login shell, `pdcs.sh` (PDC Shell). The reader may wonder why we didn't choose to use a system shell startup script. Unfortunately, the Bourne shell does not execute the system scripts in `/etc` when started as a non-login shell (C-Shell does). By the installation of a wrapper script, every process, whether started via `ssh`, kerberized `telnet/rsh` or MPI was guaranteed to be executed in our environment. Due to the design of the Easy scheduler, this modification was rather trivial to perform. Easy maintains two password files, `password.complete` and `passwd`. The former contains valid shells for all users. The latter contains valid shells only for that user who has reserved the node. This file is constructed on the fly by Easy when the job has come to the top of the queue.

The steps for job execution and finalization occur as follows:

First, a preamble script is initiated by Easy: (`pdcs-pre.sh`)

1. Check if the cluster, charge group, user and host were enabled for use with PAPI Monitoring. If not, bail out.
2. Verify the existence of the output directory tree.
3. In the above directory, create two files:
 - `BUSY`, which is a zero length file that indicates that this job is running and that monitoring is taking place.
 - `METADATA`, which contains job information that is cross referenced with that from PapiEx. It contains the following fields: cluster name, job ID, username, number of nodes reserved, charge group (CAC), start time and the finish time of the job. The finish time is filled in by the postamble script described below.

Second, Easy conditionally modifies the user's shell on all the nodes `passwd` files: (`adduser.py`)

1. Check if the cluster, charge group, user and host were enabled for use with PDCSH. If not, bail out.
2. Give the user PDC shell as his login shell on all reserved nodes.

When any job is started on any node, it will run under PDC shell and all subprocesses and threads will be monitored. (`pdcs.sh`)

1. Execute a common cluster wide setup script. (for other administrative purposes)
2. Determine the following:
 - Whether or not we are a login shell.
 - The user’s actual shell from `passwd.complete`.
3. If the cluster, charge group, user and host are enabled for PAPI Monitoring, execute the PAPI monitoring script.
4. Execute the user’s actual shell appropriately. (as a login shell or not)

The PAPI monitoring script performs the following: (`papimon.sh`)

1. Check for the file that contains the prepared arguments to PapiEx.
2. Check that these arguments are correct.
3. Verify the existence of the output directory tree.
4. Set the output environment variable to PapiEx.
5. Set up the library preload environment variables.

At this point, the user’s job runs to completion. The only processes not monitored are those that are either statically linked or they access PAPI or the hardware performance monitors directly. Upon completion of the job, a postamble runs on the front end. This script does the following: (`pdcs-post.sh`)

1. Check if the cluster, charge group, user and host were enabled for use with PAPI Monitoring. If not, bail out.
2. Append the job finish time to the METADATA file.
3. Remove BUSY file .
4. Schedule the parsing and submission of collected data to the PerfMiner database and remove/backup the original files.

3 Collecting Hardware Performance Data Transparently with PapiEx

At the lowest level, PerfMiner can use any mechanism to collect application performance data. However, other mechanisms such as requiring the user to link with a library or use custom compilation scripts are prone to user error and cause general confusion. For our setup, we wished to have a system that would be completely transparent to the user. This would require no modifications to user’s Makefiles, their application code or any dependent run-time libraries. Existing binaries would continue to run as they did prior to the deployment of the software. To accomplish this, we decided to use PapiEx, a tool based on the PAPI[PAPI]. PapiEx can run unmodified dynamically linked binaries and monitor them with PAPI. It transparently follows all spawned subprocesses and threads. In PerfMiner, the output of PapiEx is directed to a file, which is then parsed by a perl script for insertion into the database upon job completion.

4 Scalable Database Design

For the current implementation of PerfMiner system, we have chosen to use Postgres as our database back end. The primary reason for choosing Postgres was prior experience with it's operation and the support for kerberized authentication. Care has been taken to avoid the use of any nonstandard SQL that could prevent the use of Mysql, Oracle or another SQL95 compliant database. Furthermore, access to the database has been abstracted through the use of both Perl and PHP's DBI interface, providing further flexibility. Much work has been done to keep the PerfMiner database as robust and as self contained as possible. In an early implementation of PerfMiner, we rather hastily built a database schema around a common set of queries we were hoping to run. We quickly realized that this was neither general nor robust enough to support cluster wide queries spanning millions of rows. Thus a new database was designed, focusing on flexibility, extensibility and easy of implementation of sophisticated queries. Our goal was to have as much of the query processing be done by the database server itself instead of the client. Thus queries processing vast quantities of data can be done across slow links as well as performed on underpowered clients.

4.1 Direct Measurements

There are only two truly static items of knowledge in the database. First, all measurements have a target (or scope) that is one of cluster, job, node, process or thread. Secondly, there is an assumed hierarchy, that a cluster contains jobs, which contain nodes, which contain processes, which contain threads. These containments can be regarded as one to many mappings and naturally produce keys for addressing the collected data. For instance, a specific threads measurements are accessed by specifying cluster, job, node, process identifier and possibly thread identifier as the primary key. Since no assumptions of existence of any specific measurement are made, it is not possible to minimize the tables by putting all measurements of thread scope in the table that specifies which threads exist. This is, of course, unless you are prepared to accept null values and that the underlying database is able to insert columns in preexisting tables. Instead, each measurement resides in a separate table. The database also contains additional tables that describe the scope, type and meaning of each of the collected measurements. This ensures that no measurement is stored differently from any other. The primary advantage of this approach is that it makes it possible to combine measurements and construct reports in a uniform way. In PerfMiner, this means that any change in the data collected from PapiEx or from the batch system, results in the creation of a table and associated metainformation. Thus, no changes need be made to the database or to the query engine.

4.2 Derived Measurements

The measurement floating point operations per second (or FLOPS/S) is an example of a derived measurement having thread scope. It combines the direct

measurement, floating point operations, with the derived measurement, duration, which in turn is derived from clockrate and total cycles. The database is designed to store information about the derived measurements in the same way that it stores the direct measurements. The query author does not have to know if a derived or direct measurement is being referenced in his query.

4.3 Problems with the Current Approach

Putting the measurements in different tables can be perceived as discarding the fact that they are collected simulatenously and belong to the same thread. When the data is harvested, the application knows that a certain value of total cycles is associated with a certain value of total floating point operations. The only way to reconstruct this information is by joining the two tables, an $O(n^2)$ operation. This can be mitigated by instructing the database to build indexes for the fields of every metric table that serve as keys. This reduces the cost of the join to $O(n \log n)$ or less depending on the method used for indexing. However, adding indexes aggravates another problem caused by the nature of the measurements. Since the target of most measurements is threads, and the key for addressing a certain thread is made up of cluster, job, node, process, thread (of which three are TEXT-fields), the key component will strongly dominate the storage demand for most tables. A solution to this, which will be implemented, is to create synthetic keys for tables where this is a problem.

5 The PerfMiner User Interface

For the current implementation of PerfMiner, the front end runs on an Apache web server with PHP and JpGraph[JpGraph] installed. JpGraph is an open source graphing library built upon PHP and the GD library. The user is presented with a simple interface through which he can construct queries to be visualized. The resulting graph is dynamically generated with JpGraph along with a corresponding image map, such that the user can click on a corresponding portion to drill-down to more interesting data. The presentation layer of PerfMiner is arguably the area with the most potential for development. As developers, we are presented with the canonical problem of balancing functionality with interface complexity. For our initial implementation, we chose a small subset of the available data as targets of our queries. We chose to present a query interface that specified the logical and of any four items present in the job's METADATA file: four on which to scope the queries and 1 choice by which to group Cluster, Charge Group, User and Job ID. Each column is updated from the selections to it's left. Should the user choose a combination that results in the availability of a single job ID, an additional dialog is presented with the names of all the processes in that job.

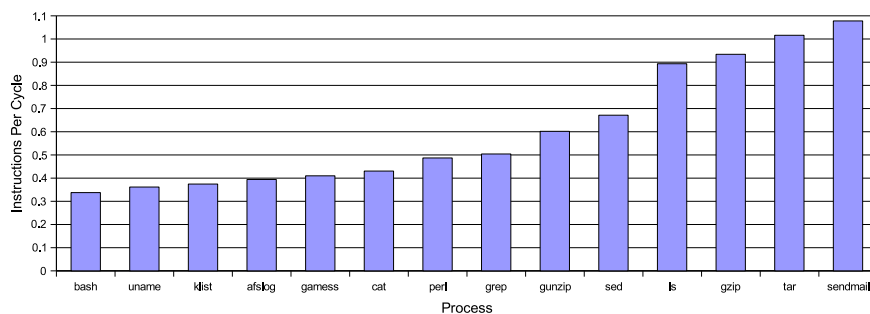


Fig. 1. PerfMiner Graph of Instructions Per Cycle of a Serial Job

6 Evaluation

The goal of PerfMiner was to be able to support three different audiences (users, system administrators and managers) through a common information collection infrastructure. For the user community, we wish to be able provide a simple way of providing performance information about recently submitted jobs without any changes to the user’s application or environment. This information could contain the efficiency of various components, the overall processing time of each component or more details hardware performance metrics. The ultimate goal is to not only provide performance information but to provide information as to why the components of that job are performing a certain way. In Figure 1, we have used PerfMiner to plot instructions per cycle against the executable name to see if we can identify which executables are running efficiently. This particular user has apparently submitted a shell script to perform a run of Gamess, an ab-initio quantum chemistry package. Here we find that Gamess was far from the most efficient executable in this particular job.

For the administrator and support staff community, we may not be so interested in actual process performance, but rather the throughput of the system and the users’ jobs. In Figure 2, we have chosen a query to find out which users of our Itanium 2 cluster performance of the system as a whole. For this particular query, we have asked the system to plot the average level 1 data cache hit rate of jobs and sort the results by user. Not surprisingly, the user with the highest miss rate is utilizing the most compute cycles. This kind of query is extremely powerful when aiming to maximize the throughput of a particular system. It’s not hard to envision a scenario where application specialists approach the user in question and offer help to optimize the user’s code.

Lastly, PerfMiner’s goal is to be able to facilitate a good understanding of exactly how the systems are being used by the various user communities. By doing so, they can plan appropriately for future procurements. The central idea here

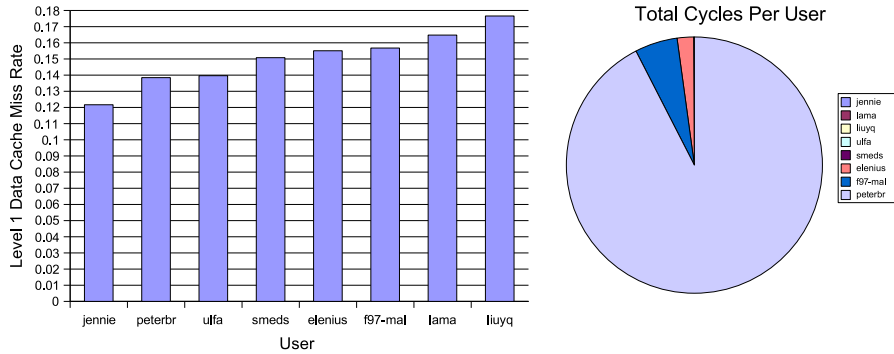


Fig. 2. PerfMiner Graphs of Level 1 Data Cache Miss Rates and Total Cycle Consumption by User

being that they can focus their procurements on having the type of hardware appropriate for the problems being solved. Should the user workload demonstrate high cache hit-rates and counts of floating point instructions, perhaps a system with a similar size cache but a higher core clock frequency and deeper floating point pipeline would be an appropriate upgrade. Should the workload demonstrate low processor utilization and low TLB-miss rates, perhaps an upgrade of the I/O subsystem would be more appropriate than a processor upgrade. The key here is to remove the guesswork involved in the procurement process. Instead of focusing next generation purchases on either artificial benchmark suites or a select group of applications, the procurement could be based on exactly what the user community has demonstrated a need for.

7 Related Work

There are numerous systems in existence that do cluster-wide performance monitoring. Many of them like Ganglia[Ganglia], SuperMon[SuperMon], CluMon[CluMon], NWPerf[NWPerf] and SGI's Performance CoPilot[PCP] are extensible frameworks capable of just about any metric. All these systems gather their metrics only on a system wide basis through a daemon process that scrapes the `/proc` filesystem. PerfMiner is most closely related to (and inspired by) the pioneering work done by Rick Kufirin et al at the National Center for Supercomputing Applications[Kufirin1]. In that work, a locally developed PAPI based tool PerfSuite[PerfSuite] is used to collect information on jobs sent through their batch system. The primary differences between our work are the collection mechanism, the design of the database and the user interface. Through personal correspondence with the authors we have learned that there is an internal Web interface through which they make queries, but that form and function of interface remains unclear. In contrast, PerfMiner's audience and usage model drives the implementation, with rapid deployment, ease of use and clarity of the presented data being our primary objectives.

References

- [SvPablo] Reed, D. A., et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. Proc. Scalable Parallel Libraries Conf. IEEE Computer Society. (1993) 104–113
- [Easy] Lifka, D., Henderson, M., Rayl, K.: Users guide to the argonne sp scheduling system. Technical Report **ANL/MCS-TM-201** (1995)
- [ParadyN] Miller, B. et al. The ParadyN Parallel Performance Measurement Tool. IEEE Computer **28/11** (1995) 37–46
- [TAU] Mohr, B., Malony, A., Cuny, J.: TAU Tuning and Analysis Utilities for Portable Parallel Programming. Parallel Programming using C++, M.I.T. Press. (1996)
- [PAPI] Mucci, P. et al. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. Proceedings of Supercomputing 2000. (2000)
- [JpGraph] Persson, J. JpGraph - OO Graph Library for PHP. <http://www.aditus.nu/jpgraph/index.php>
- [PerfSuite] Kufrin, R. The PerfSuite Collection of Performance Analysis Tools. <http://perfsuite.ncsa.uiuc.edu>
- [Ganglia] The Ganglia Scalable Distributed Monitoring System. <http://ganglia.sourceforge.net>
- [PCP] Performance Co-Pilot <http://oss.sgi.com/projects/pcp>
- [SuperMon] SuperMon High Performance Cluster Monitoring. <http://supermon.sourceforge.net>
- [CluMon] Fullop, J. CluMon Cluster Monitoring System. <http://clumon.ncsa.uiuc.edu>
- [NWPerf] Mooney, R. et al. NWPerf: A System Wide Performance Monitoring Tool Poster Session 31, Supercomputing 2004, Pittsburg, PA.
- [Petrini] Petrini, F. et al. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q Proceedings of Supercomputing 2003. (2003)
- [Mucci] Mucci, P. et al. Application Performance Analysis Tools for Linux Clusters. Linux Clusters: The HPC Revolution 2004, Austin, TX. (2004)
- [Kufrin1] Kufrin, R. et al. Automating the Large-Scale Collection and Analysis of Performance Data on Linux Clusters Linux Clusters: The HPC Revolution 2004, Austin, TX. (2004)
- [Kufrin2] Kufrin, R. et al. Performance Monitoring/Analysis of Overall Job Mix on LargeScale Pentium and Itanium Linux Clusters SIAM Parallel Processing, San Francisco, CA. (2004)
- [Monitor] Mucci, P., Tallent, N. Monitor - user callbacks for library, process and thread initialization/creation/destruction. <http://www.cs.utk.edu/~mucci/monitor>