Autoperf: Workflow Support for Performance Experiments

Xiaoguang Dai
University of Oregon
xdai@uoregon.edu

Boyana Norris
University of Oregon
norris@cs.uoregon.edu

Allen D. Malony
University of Oregon
malony@cs.uoregon.edu

ABSTRACT

Many excellent open-source and commercial tools enable the detailed measurement of the performance attributes of applications. However, the process of collecting measurement data and analyzing it remains effort-intensive because of differences in tool interfaces and architectures. Furthermore, insufficient standards and automation may result in losing information about experiments, which may in turn lead to misinterpretation of the data and analysis results. Autoperf aims to support the entire workflow in performance measurement and analysis in a uniform and portable fashion, enabling both better productivity through automation of data collection and analysis and experiment reproducibility.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Tools, Performance

Keywords

performance measurement, performance analysis

1. INTRODUCTION

The typical workflow of performance analysis involves several steps. At the beginning, one must decide on the particular approach and tool for generating the performance data, which can be achieved through instrumentation (from source level to binary level) or sampling. Tools such as PDT [14] provide source-level instrumentation, while MAQAO [8] and Dyninst [4] can be used for binary instrumentation. TAU [17] and HPCToolkit [1] are two of the best open-source tools that can generate performance data for users. TAU supports both instrumentation and sampling, but HPCToolkit is designed only for sampling.

Next, one must decide what to measure. In most cases, this involves using hardware performance counters, or metrics derived from them. PAPI [5] is widely used as an abstraction layer to collect those counters in a (mostly) architecture-independent manner. In practice, however, due to hardware restrictions (e.g., on Intel platforms), we cannot measure all metrics simultaneously, either because there are not enough PAPI-supported hardware counters, or some of the counters cannot be grouped together for measurement. In the latter case, metrics must be partitioned manually to execute the same performance experiment multiple times, with partial data collected in each iteration.

Now it is time to run the experiment. The exact way to do so depends on the environment being used. For example, a job control system may or may not be necessary, and different computation environments can use different job control systems. Moreover, some common tools (e.g. *mpiexec* for MPI applications) depend on special (non-standard) arguments to function properly or require a background process to be running (e.g., *mpd*).

Next, data collected from each iteration of the experiment must be aggregated for subsequent analysis. Two common analysis tasks are to calculate derived metrics and to compare two distinct experiments to find correlations. The open-source PerfExplorer [12] framework works with TAU and provides many useful analyses. It also exports a Jython API to enable scripting, but requires some training to learn how to use effectively.

In practice, the workflow listed above is tedious and error-prone. Partitioning metrics manually is laborious, even with the help of `papi_event_chooser` and must be repeated for each different architecture. Writing and debugging scripts for each iteration is time consuming. Running in a different computational environment requires porting of the measurement infrastructure and repetition of some of the steps (e.g., selecting metrics and grouping of counters). General performance tools don't provide consistent, easy-to-use, yet flexible interfaces to support specific analysis requirements on a variety of platforms.

In this paper we introduce the Autoperf tool for creating and managing performance experiments, including data postprocessing and analysis [3]. The main contributions of this work are (1) the automation of previously manual and error-prone portions of the performance measurement and analysis workflow; (2) simple experiment configuration specification that encapsulates sufficient information to automate the execution of performance experiments and subsequent data analysis; and (3) a modular implementation that allows any part of the workflow to be extended, e.g., by integrating new performance measurement tools or analyses. These are the first steps toward addressing the currently unsolved challenge of *efficiently* collecting and analyzing fine-grained application performance data on diverse platforms supporting different sets of performance measurement tools in a *reliable* and *reproducible* manner.

1.1 Related Work

PerfExpert [6] is a tool whose goal is to automatically diagnose core, socket, and node performance bottlenecks in parallel HPC applications. The user interface is intentionally simple. Similar to Autoperf, PerfExpert relies on other tools to perform the actual measurements but to our knowledge is limited to sampling-only data collection (through HPCToolkit and PAPI). While some of the goals of Autoperf are similar, our approach focuses on ease of extensibility (to encode and share expert knowledge), consistent experiment representation across platforms and applications, ultimately leading to reproducible (and reusable) experiments. Autoperf is also tool-agnostic and can be interfaced with different measurement and analysis tools.

A number of commercial tools provide some experiment management, and various analysis and visualization capabilities, including Intel's VTune [13], Vampir [15], Cray's CrayPAT [7], NVIDIA's Nsight [16], and ThreadSpotter[1] [9]. These tools offer many valuable capabilities but are limited to specific platforms, are closed source, or cannot be easily extended.

1.2 Autoperf

Autoperf provides a simple format for defining the experiment environment and data to be collected, and interfaces to TAU, PAPI, HPCToolkit, and PerfExplorer to perform the measurements and subsequent analyses. The current capabilities include the collection of detailed hardware performance counters, derived performance metrics computations, statistical analysis, and preliminary support for comparisons of different code versions.

Autoperf is implemented as an extensible, modular Python package that significantly automates performance experiments. For data collection and some of the analysis, Autoperf leverages other tools, adding automation for the portions of the performance measurement and analysis workflow that are not supported by the underlying tools. The ultimate usability goals are to (1) reduce or eliminate manual work for users; (2) to enable analyses to be saved, reused, and extended; and (3) to provide the same interface through which multiple performance tools can be used.

2. DESIGN

Autoperf comprises four components: experiment specification, job submission, data collection, and analysis engine, implemented as independent Python modules.

2.1 Experiment Specification

Some basic information is required before Autoperf can run the experiment, e.g., the commands used to invoke the application, where to find other dependent tools, and which metrics to collect. The metrics specification is one of the types of inputs that can be further abstracted in the future, but at the moment users can select metrics they are interested in, or use one of a number of examples included in Autoperf as a template for their own experiments.

[1]ThreadSpotter is now available as open source and a great target for integration with Autoperf.

Instead of relying on a lot of information passed through the command line or environment variables, Autoperf reads the experiment specification from a configuration file. Using files makes the sharing and reuse of configurations easy, enables change tracking, collecting and recording of all relevant provenance data at the time the experiment was performed. The format of the configuration file resembles a classic Windows INI file (and uses the Python builtin ConfigParser module), with additional hierarchy semantics between sections defined by Autoperf, which we illustrate with the following example using the miniFE Mantevo mini app [10].

```
[Main]
  Experiments = miniFE_20 miniFE_50
[Experiments]
  execmd = ./miniFE
[Experiments.miniFE_20]
  exeopt = nx=20
[Experiments.miniFE_50]
  exeopt = nx=50
```

Two miniFE experiments are defined in the `[Main]` section. Experiment names can be arbitrary strings and can optionally include Python code segments defining certain Autoperf keywords to allow more flexible configuration settings (e.g., `miniFE_20 {threads = [1] + range(2,9,2)}` will result in five experiments with 1,2,4,6, and 8 threads). The command used to run the application, in this case `miniFE`, is specified in the `[Experiments]` section. However, in order to provide different arguments to `miniFE_20` and `miniFE_50`, we extend the `[Experiments]` section by adding the corresponding experiment name as a postfix to create a derived section and list experiment-specific options there. Options that are redefined in a derived section override those appearing in the parent sections. The same approach applies to other configuration file sections, not just to `[Experiments]`, keeping the configuration file concise, expressive, and reusable.

Many options can be specified when setting up the experiment environment, which can add to the manual effort required to create experiments. Autoperf attempts to minimize this effort by providing reasonable defaults for most of the environemnt setup and also some limited Python script embedding. Autoperf composes the `execmd` and `exeopt` accordingly so that the application can be launched in a desired manner. For instance, if "`mpi = yes`" is specified in the configuration file, Autoperf will use the MPI launcher (`mpirun` or `mpiexec`) to run the testing application. When using TAU for sampling-based measurement, Autoperf calls `tau_exec` with the correct options and environment settings. Complex configurations that cannot be specified with a single option are grouped together to create a new standalone section.

2.2 Job Submission

Many HPC systems use a job scheduler such as PBS to allocate and manage system resources. Users are typically required to write a script and submit it to the job scheduler, which allocates nodes and other necessary resources and then launches the application. Clusters may also have some specific local configuration requirements, which need special treatment. For example, on the University of Oregon ACISS cluster, the `--mca btl_tcp_if_include torbr` command-line option must be specified for `mpiexec` in order to launch a MPI application. These platform-dependent settings are abstracted in the `Platform` configuration section. Each `Platform` option value is implemented as a separate Python module, which handles environment-specific details. The `Queue` option itself has a number of settings, so it is described in its own (optional) subsection, `[Queue]`.

```
[Experiments]
  Platform = aciss
[Platform]
  Queue = PBS
[Queue]
  nodes     = 2
  ppn       = 12
  walltime  = 4:00:00
```

The above sections specify that the experiment will run on ACISS, the job will be submitted by PBS, and required computation resources are specified in the `[Queue]` section. Autoperf then generates all required scripts and submits the job. Autoperf also provides a way to examine the status of running jobs that are part of specific experiments.

2.3 Data Collection

Autoperf relies on existing performance tools to collect performance data. Currently, TAU and HPCToolkit are supported, and others can be added relatively easily. All collected data are converted to TAU PPK format.

TAU supports source-level instrumentation which requires recompilation of the application. Providing general support for the build process of arbitrary applications is a complex task, hence this part is left to users at present. However, we provide a hook in the configuration file, allowing an arbitrary user provided command to be executed before running the experiment. The example below shows the use of this `builder` option:

```
[Experiments]
  builder = make -C .. miniFE
```

One major prerequisite for running a performance experiment manually is the selection of hardware counter-based metrics that can be measured simultaneously. Autoperf eliminates this tedious task by automatically partitioning both PAPI and the NVIDIA CUDA Profiling Interface (CUPTI) counters. As we show in the next section, after a list of counters for analysis is specified, Autoperf checks whether they can be measured together. If not, Autoperf figures out how to partition the counters into compatible groups and run a sub-experiment for each of them. When all the sub-experiments finish, collected partial data is aggregated to create the complete result of the original experiment.

Another typical workflow step is to calculate derived metrics. Autoperf resolves dependencies among metrics to measure all quantities required for specific derived metrics computations. By adding metric specifications (i.e., mathematical expressions used to calculate derived metrics), into the metric database folder, derived metrics can be used as raw data metrics. Furthermore, derived metrics can refer to each other in their specifications. The sample metric specification below shows how to use several PAPI counters and system-specific metadata (e.g., CPU frequency or other architectural parameters determined through microbenchmarks) to compute a derived metric, floating-point inefficiency.

```
$ cat util/metric_spec/FP_INEFFICIENT2
  ((PAPI_FP_INS/PAPI_TOT_INS)*(PAPI_RES_STL/
  PAPI_TOT_CYC))*(PAPI_TOT_CYC/META_CPU_HZ)
```

Should saving performance data into permanent storage for later use be preferable, TAUdb [11] provides such a facility. Data can be stored in a shared TAUdb database or a user-created local database by running `taudb_configure` to create a TAUdb configuration file that specifies how to connect to the database. After data are collected, Autoperf will load data into the database automatically according to its specified configuration file. Users do not need to know how to use databases.

```
[Experiments]
  Datastore = taudb
[Datastore]
  config = my_taudb_config_name
```

Several metadata are saved when collected performance data are loaded into the database. The loaded data can be queried using the metadata, or checked out using an unique instance id, which is its timestamp.

2.4 Analysis Engine

Autoperf provides two interfaces to support customized analysis on performance data.

2.4.1 PerfExplorer Script

Because performance data are converted to TAU PPK format and loaded into TAUdb, the Java-based PerfExplorer framework is a natural tool choice for providing a number of analyses. When the PerfExplorer builtin functionality is insufficient, one option is to write a customized Jython script that uses the PerfExplorer analysis API.

Autoperf provides easy interfaces to several PerfExplorer scripts (and we continuously add more). For example, Autoperf can find collected data specified in the experiment configuration, and retrieve it from the TAUdb database if it is not available on the local disk. Autoperf also helps transform raw data into the PerfExplorer API internal data structures, enabling users to analyze data directly without requiring a database connection.

The sample below shows how Autoperf compares `miniFE_50` with the latest instance of `miniFE_20` (see Fig. 1). The comparison finds the top 10 methods that produce the biggest difference for the listed metrics, and then save the result summary in a text file and generate a histogram chart for each of the listed metrics. Autoperf hides the details of where or how to retrieve the latest data for `miniFE_20`. This example also illustrates that a derived metric can be used along with regular metrics indistinguishably providing that the metric is properly specified in the experiment configuration. While here we compare experiments with different data inputs, the same method can be used to compare different implementations based on an unlimited, extensible set of metrics of interest.

```
[Experiments.miniFE_50]
  Analyses = compare2
[Analyses.compare2.miniFE_50]
  base      = miniFE_20
  instance  = last
  threshold = 10
```

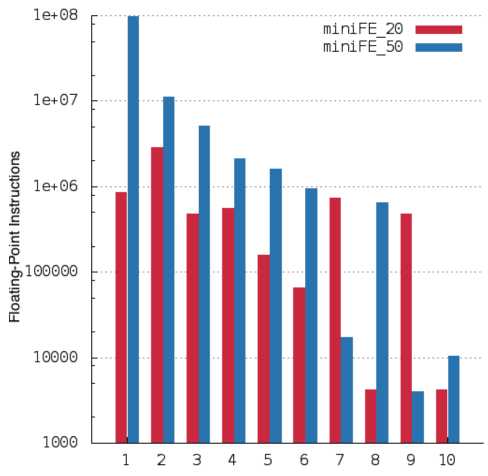

Figure 1: A visual summary of the Mantevo experiment example showing floating-point instructions (y-axis) for the top 10 functions for each experiment (w.r.t. largest difference in mean floating-point instructions per function).

```
metrics   = PAPI_FP_INS FP_INEFFICIENT2
```

Many existing PerfExplorer scripts can be merged into the Autoperf framework without any change.

2.4.2 SciPy

All collected data are converted to PPK format. Instead of always depending on PerfExplorer to load data from PPK archive, Autoperfhas its own PPK parser. This provides the same set of capabilities as the PerfExplorer scripting interface, plus several other advantages.

First, using NumPy arrays for all data enables the use of packages such as SciPy or scikit-learn. Second, PerfExplorer scripts require explicit API calls to the Java implementation to calculate each derived metric. By contrast, the PPK parser interprets derived metric specification and applies it automatically to the data. Hence, derived metrics are calculated and populated without extra effort from users or the JVM overhead. Finally, one inconvenience of sampling compared to instrumentation is that samples may occur in third party libraries for which no function symbols are available, resulting in data that is of little use without further processing. Autoperf unrolls the call stack and backtraces the call path to the first method that is of interest, and then attributes collected data to this method. This ensures hotspots in the code can be traced.

3. AUTOPERF WORKFLOW

The first step in using Autoperf is to write a configuration file which specifies following:

- How to find and run the application (executable, command-line arguments, symbolic links);
- The environment used to run the application (e.g., PBS);
- The tool to use for data collection (e.g., TAU);
- The analysis you want to apply on collected data (e.g., correlation).

The complete example is shown below.

```
[Main]
  Experiments = miniFE_20
                miniFE_50

[Experiments]
  rootdir    = output
  tauroot    = ~/tau/x86_64
  Platform   = aciss
  Tool       = tau
  Datastore  = taudb
  mpi        = yes
  execmd     = ./miniFE.x

[Experiments.miniFE_20]
  exeopt     = nx=20
  Analyses   = metrics

[Experiments.miniFE_50]
  exeopt     = nx=50
  Analyses   = compare2
[Platform]
  Queue      = PBS

[Datastore]
  config     = demo

[Tool.tau]
  mode        = sampling
  TAU_EBS_UNWIND = 1

[Analyses.metrics]
  metrics = PAPI_FP_INS
            PAPI_L1_DCM

[Analyses.compare2]
  metrics    = PAPI_FP_INS
               FP_INEFFICIENT2
  base       = miniFE_20
  instance   = last
  mode       = absolute
  throttle   = 1000
  threshold  = 10
```

Next, the user launches the driver script `autoperf` to run all the experiments included in the configuration file

```
$ autoperf -f <cfg_file>
```

A simlpe 20-line configuration file can produce tens or hundreds of individual jobs, which are generated, tracked, and processed automatically by Autoperf. The following command can be used to check the status of experiments:

```
 $ autoperf -c
Experiment  Instance                      Job ID          Status
 --------------------------------------------------------------
miniFE_20   2014-11-04-15-40-16-666930  PBS:558834.hn1  Running
miniFE_50   2014-11-04-15-40-19-478332  PBS:558835.hn1  Running
```

After the experiment has finished running, the user can analyze the results with the "`autoperf -y`" command. To combine the measurement and analysis steps, Autoperfcan be invoked with "`autoperf -b`". In this case, Autoperfwill submit the job(s), blocking until they complete, and then performing the analysis on the results.

4. EXAMPLE USE CASE

We demonstrate some of the Autoperf capabilities by analyzing a test application developed for the Geant4 vector prototype. Geant4 [2] is a toolkit for the simulation of the passage of particles through matter, which is widely used in high energy physics (it is the reference simulation engine for LHC experiments at CERN and other HEP labs), medicine, and other application areas. The vector prototype is one of the ongoing parallelization efforts in the Geant4 community whose goal is to create an implementation that performs well on both modern CPUs and hybrid systems including accelerators such as the Intel Xeon Phi.The code incorporates explicit vector instructions.

We ran all experiments on the University of Oregon ACISS cluster (each node is a 12-core Xeon X5650@2.67GHz with 72GB DRAM), using 4 threads. The code was compiled with GCC version 4.8.3.

Table 1 lists the top five functions sorted by total number of cycles in the unoptimized (compiled with `-O0`) version. The wall-clock times of the different versions were approximately 5, 2.25, and 2.3 minutes for `-O0`, `-O2`, and `-O3`, respectively.

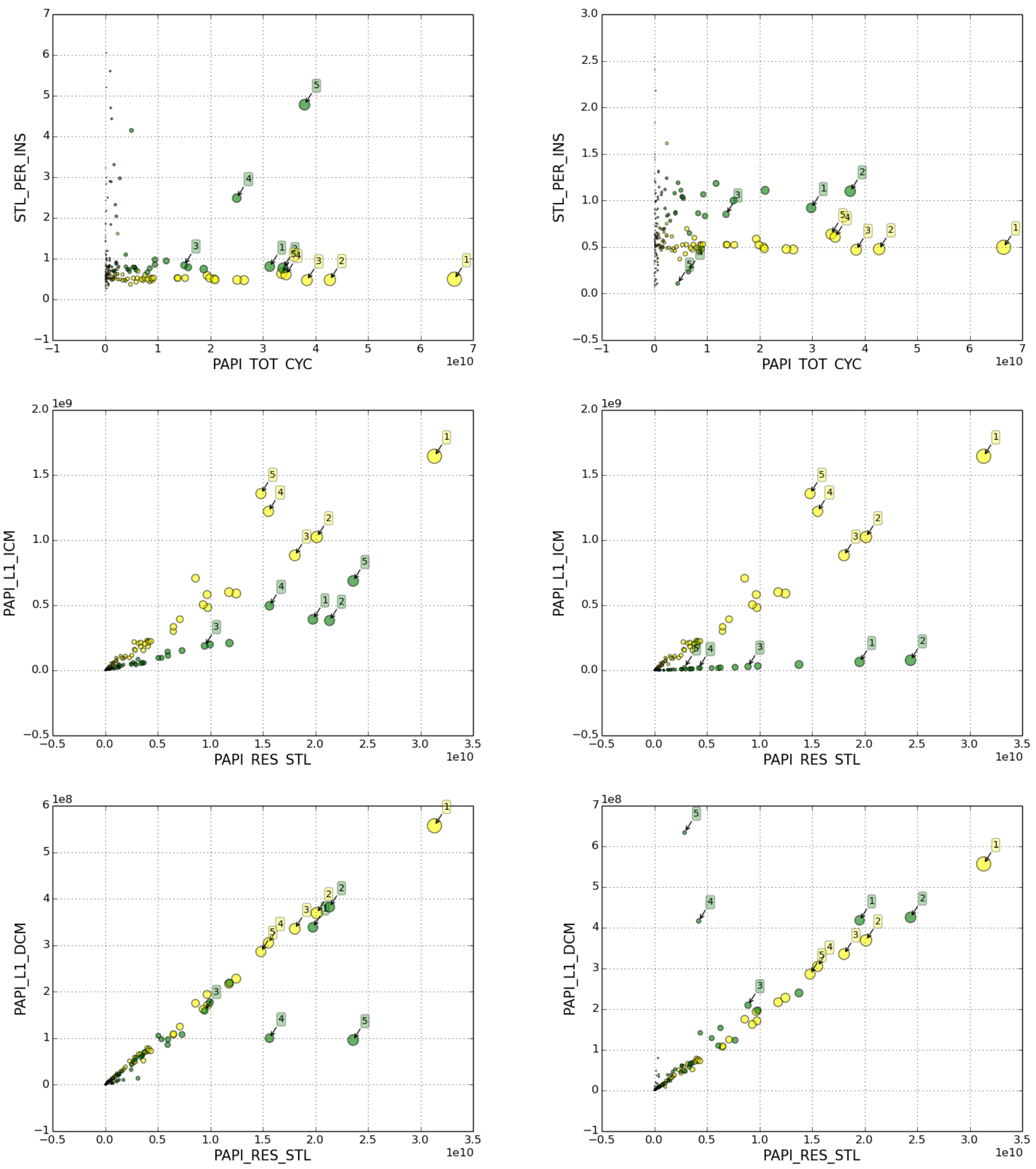

Figure 2: Each bubble corresponds to a function. The left column corresponds to code compiled with the `-O2` compiler flag (green bubbles), while the right column shows the same metrics for `-O3` (green bubbles). The same baseline `-O0` version is shown in yellow. Each bubble's diameter is proportional to the percentage of total cycles for each function. The top five (yellow) functions (in `-O0`, Table 1) and their optimized (green) versions are labeled $1, 2, 3, 4, 5$ (see Table 1). The rows are: (1) stalls per instruction vs total cycles—O2 unexpectedly increases stalls per instruction in two of the functions; (2) level-1 instruction cache misses vs stall cycles—O3 eliminates most instruction misses, but with little impact on stalls, showing that they are not a major contributor to stalls; (3) level-1 data cache misses vs stall cycles—O3 increases L1 misses in some functions, but with little effect on overall stalls.

Table 1: Top five functions (total cycles)

Stall Cycles ($\times 10^{10}$)	Total Cycles ($\times 10^{10}$)	Function Name
3.13	6.65	GeantTrack_v::ComputeTransportLength
2.01	4.28	GeantTrack_v::PropagateInVolumeSingle
1.80	3.84	WorkloadManager::TransportTracks
1.55	3.44	GeantTrack_v::AddTracks
1.48	3.36	TOPLEVEL

Figure 2 illustrates some of the automatically generated results. While this example is a comparison between versions compiled with different optimization options, in general the same analyses can be performed for any distinct code versions, whether manually created (e.g., comparing with an older version of the same application or an alternate experimental branch) or synthesized with a tool (compiler, auto tuner), the same analysis capabilities are available and the Autoperf configuration file is largely the same.

5. CONCLUSIONS AND FUTURE WORK

We have introduced the Autoperf tool for managing performance experiments by providing consistent interfaces to common measurement tools and automating time-consuming and error-prone portions of the performance analysis workflow. The tool is in the early stages of development and we will be expanding it with more reusable analysis modules and platform configurations. Thanks to the the ease of performance data gathering and analysis, large quantities of data can be generated quickly. We will extend the current simple correlation analysis to enable the automated classification of *significant* metrics (raw hardware counters or derived metrics), i.e., estimating the sensitivity of a particular response (e.g., execution time) to changes in application behavior. We will also add support for more performance measurement and analysis tools (e.g., ThreadSpotter) as well as modeling interfaces (e.g., Descartes Query Language).

Acknowledgments

We are grateful to Kevin Huck of University of Oregon for his continuous help with TAU and PerfExplorer, and valuable feedback during the design and implementation of Autoperf. This research is supported by the DOE SciDAC grant DE-SC0006723.

6. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, Apr. 2010.

[2] J. Allison. Geant4 developments and applications. *IEEE Trans. on Nuclear Science*, 53(1):270–278, 2006.

[3] Autoperf. `https://bitbucket.org/xdai/autoperf`.

[4] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proc. of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16. ACM, 2011.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[6] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[7] Cray. Optimizing Applications on the Cray X1TM System. `http://docs.cray.com/books/S-2315-50`.

[8] L. Djoudi and D. Barthou. Maqao: Modular assembler quality analyzer and optimizer for Itanium 2. *Workshop on EPIC architectures and compiler technology*, 2005.

[9] E. Hagersten. ThreadSpotter (Rogue Wave Software). `http://www.vi-hps.org/upload/projects/hopsa/hopsa-nov12-threadspotter.pdf`.

[10] M. A. Heroux, D. W. Doerer, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Sept. 2009.

[11] K. Huck, A. Malony, R. Bell, and A. Morris. Design and implementation of a parallel performance data management framework. In *ICPP2005*. IEEE Computer Society, 2005.

[12] K. A. Huck and A. D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Proc. of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 41–. IEEE Computer Society, 2005.

[13] Intel. VTune Performance Analyzer. `http://software.intel.com/en-us/intel-vtune/`.

[14] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000.

[15] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[16] NVIDIA. NVIDIA Visual Profiler. `https://developer.nvidia.com/nvidia-visual-profiler`.

[17] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.