A Toolchain for Profiling Virtual Machines

Jiaqi Zhao
School of Basic Sciences
Changchun University of Technology, China
scorpiozhao@yahoo.com.cn

Jie Tao
Steinbuch Center for Computing
Karlsruhe Institute of Technology, Germany
jie.tao@kit.edu

Lizhe Wang
Center for Earth Observation and Digital Earth
Chinese Academy of Sciences, China
lizhe.wang@gmail.com

Andreas Wirooks
Steinbuch Center for Computing
Karlsruhe Institute of Technology, Germany
andreas.wirooks@kit.edu

ABSTRACT

Performance tuning is a common topic in the research domain High Performance Computing. Currently, various tools have been developed to help programmers understand the runtime execution behavior of their applications. It is clear that such tools are also required for performance analysis on virtual machines, where applications, together with their execution environment, sit on top of a virtualization layer rather than running directly on the physical machines.

This work developed a toolchain (also called workflow system in the following), specifically for performance analysis on virtual machines. Starting with a profiling tool, the workflow system first collects the runtime performance data on both physical and virtual machines. The performance data are filtered, combined, transformed, and then delivered to a visualization tool, where graphical views are produced to demonstrate the performance difference between native executions and the execution on virtual machines. We tested the toolchain with standard benchmark applications running either sequentially or in parallel with multiple threads.

KEYWORDS

Performance Tools, Profiling, Virtual Machines, Performance Analysis

INTRODUCTION

Virtual machines have been increasingly used in different scientific domains for various purposes like provisioning customized computing environments, running legacy codes, fault-tolerance, and easy system management. However, virtualization causes a performance loss due to the fact that applications now run on top of a virtualization layer, the so-called hypervisor or Virtual Machine Monitor (VMM). A tool support for comparative study of the performance and the runtime execution behavior of applications on both physical and virtual machines can surely help programmers find reasons that cause the performance deficit on virtualized architectures and further optimize their applications towards a performance improvement.

Actually, performance tools have long been applied by programmers in performance analysis and tuning. Over the last years, a number of performance tools have been implemented. Vampire (Brunst et al., 2009), TAU (Shende et al., 2006), and Intel Vtune (Intel, 2013) are several well-known and widely used examples. These tools have significantly supported programmers in developing scientific codes (Ciorba et al., 2010; Malony et al., 2011; Hammond et al., 2012).

Existing tools are mostly developed for applications running directly on the host machines. For virtual machines, however, there are few implementation work that supports performance analysis. Observing available performance tools it can be seen that the role of a performance tool is to present the runtime behavior of events like memory access, cache access, inter-process communication, synchronization, and I/O operations. This role remains for virtual machines. However, an additional point has to be considered for virtual machine specific tools, i.e., the difference between physical and virtual machines for the same event. Programmers have to compare the performance of executions on virtual machines with the physical runs to detect bottlenecks, where physical machine outperform the virtual ones. This discovery is the base for any further optimization. Therefore, providing a comparative view of the performance events must be a major focus of tools specifically designed for virtual machines.

We implemented a toolchain, a workflow system, that builds an automatic process, from data acquisition up to visualization, to support programmers in the task of performance analysis on virtualized architectures. The runtime performance data are collected using the Linux profiling tool *perf*, which is delivered together with the Linux kernel. Performance data on both the physical machine and the virtual machine are then filtered, combined, and converted to a specific form that can be handled by BIRT (Eclipse, 2013), an Eclipse-based open source reporting system actually designed for Web applications. In this work we use BIRT as a visualization tool to illustrate the performance data with graphical presentations. We validated the developed toolchain with OpenMP applications from standard benchmark suites,

Proceedings 27th European Conference on Modelling and Simulation ©ECMS Webjørn Rekdalsbakken, Robin T. Bye, Houxiang Zhang (Editors)
ISBN: 978-0-9564944-6-7 / ISBN: 978-0-9564944-7-4 (CD)

running in both the sequential mode or parallel with multiple threads. Experimental results show some interesting behavior on the virtual machines.

The remainder of the paper is organized as follows: Section gives an overview of profiling on virtual machines, together with some related work. Section describes the developed workflow system in detail. Section shows sample experimental results. The paper concludes in Section with a short summary and several future directions.

PROFILING VIRTUAL MACHINES AND THE RELATED WORK

The runtime performance data are the basis for any performance analysis tools. Such data can be collected by different software approaches like simulation and code instrumentation. However, a more direct way to gather the runtime information about the applications memory access or execution behavior is to use the hardware counters provided by modern processor architectures in the form of a Performance Monitoring Unit (PMU). These counters can record the occurrence of various CPU or software events such as cache miss, TLB miss, I/O events, and specific instructions.

In the last years, different low-level libraries or high-level interfaces have been implemented for programmers to access the performance counters in the source codes. *perf* (de Melo, 2010) is a low level profiling tool delivered by the Linux kernel. It provides a simple commandline interface for uses to start profiling a running application for specific events. *perf* supports a number of measurable events including the software events, like context-switches and minor-faults, as well as the hardware events such as the number of CPU cycles and cache misses. *perf* aggregates the occurrences of the user-specified events and provides the profiling results as perf-reports at the end of applications execution. The results with the events are split into individual functions including the routines of system libraries.

PAPI (Browne et al., 2000) is a well-known and widely applied programming interface for accessing the hardware counters within an application's source code. PAPI provides both a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface to the underlying counter hardware. The low level PAPI interface deals with hardware events in groups while the high level interface simply provides the ability to start, stop, and read specific events. A set of performance tools, including TAU and Vampire, use PAPI as the interface to performance counters. Besides PAPI, there are other well-known programming interfaces for hardware counters. *perfmon* (Sourceforge, 2013) and oProfile (OProfile, 2013) are two system-wide profilers for Linux systems.

On virtual machines, however, the hardware counters are invisible to the applications, i.e., they cannot be directly accessed within a virtual machine. The access to these registers can only be done via the hypervisor. Xenoprof (Menon et al., 2005) is an implementation that allows the performance profiling within a running virtual machine instance. It is a system-wide statistical profiling toolkit implemented for the Xen virtual machine environment. The Xenoprof toolkit supports coordinated profiling in a Xen environment to obtain the distribution of hardware events. Xenoprof allows the profiling of concurrently executing virtual machines, and provides profiling data at the fine granularity of individual processes and routines executing either within the virtual machine or on the physical host.

Researchers have applied the Xenoprof toolkit to study application performance on architectures virtualized with the Xen hypervisor. For example, work in (Youseff et al., 2008) studies memory hierarchy features of para-virtualized machines and memory intensive applications; work in (Tikotekar et al., 2008) studies the behavior of L2 cache, DTLB, ITLB, and the overall performance penalty of HPC applications.

Xen (Barham et al., 2003) is a well-known hypervisor and an open source development that has been widely used for system virtualization. It supports both para-virtualization and full virtualization, where the former uses hypercalls for the guest Operating System (OS) to communicate with the hypervisor with the necessarily of a slight OS modification, while the latter translates the sensitive instructions to a new sequence of instructions for the virtualized hardware without changing the OS running on a virtual machine.

Nevertheless, there is currently a trend of replacing Xen with KVM (Kernel-based Virtual Machine) (KVM, 2013). KVM is also an open source product. It adds the virtualization capacities directly in the Linux kernel, achieving the thinnest hypervisor of only a few hundred thousand lines of code. Therefore, KVM is being increasingly deployed, even though it requires the virtualization extensions in the hardware of modern microprocessors (e.g. Intel VT and AMD-V).

For KVM-virtualized machines, however, there exists no profiling tool, similar to Xenoprof, which supports system-wide profiling also within the virtual machines. Fortunately, KVM developed recently a virtualized performance monitoring unit (VPMU) that emulates the hardware counters and provides each virtual machine with a performance monitoring unit. The Linux system tool *perf* already supports VPMUs and can be used for system-wide profiling in KVM-based systems. Some high level interfaces, like PAPI, also supports now the VPMUs. However, they currently run only on Intel processors. AMD architectures have not been supported yet.

We aim at profiling KVM-virtualized machines, independent of the target architecture. Therefore, we use the low level Linux tool *perf* to collect initial performance data. For a better understanding of the counter events as well as a comparison between performance on the host and the virtual machines, we developed the workflow system of tools to visualize the runtime behavior of sys-

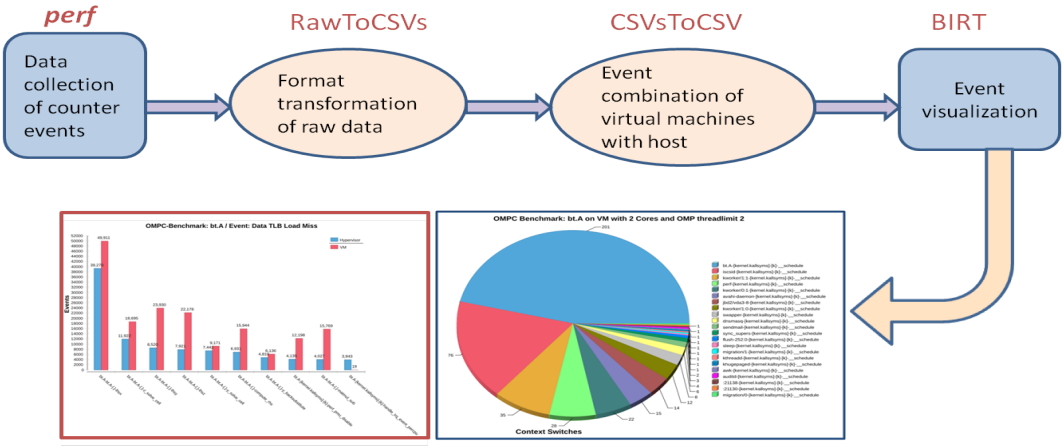

Figure 1: The toolchain for collecting, transforming, and visualizing event data.

tem hardware and software events.

THE WORKFLOW SYSTEM

The developed workflow system is illustrated in Figure 1. The whole toolset is based on the *perf* tool that can be used to collect initial performance data from hardware counters. *perf* offers a simple commandline interface for users to interact with the system. During the profiling period *perf* writes the events with their related memory addresses in a binary file, which can be transformed to a text report after the profiling.

perf can be run in several modes. Mode *stat* allows the users to profile an application as a whole, whereby only events that occur within the application are counted together without a detail insight into the individual methods of both the program and the system library. Mode *record* enables the system-wide profiling and is based on sampling. Hence, this mode is not accurate in contrast to *stat*. However, with this mode it is capable of acquiring a detailed report about the methods of a program as well as the libraries and kernel functions at either thread or CPU level. Mode *top* performs also system-wide profiling but shows the results directly on the terminals like the standard Unix command “top” rather than recording the events in a file. *kvm* is a special mode that works only for a KVM-virtualized machine, where the host and several virtual machines run in parallel. It functions similarly to *record* and works on both the host and a virtual machine. This mode is exactly what we need for building the toolchain.

However, the *perf* report, even in a text format, is a low-level presentation. Therefore, we follow the traditional solution to show the performance data graphically with a visualization tool, the Business Intelligence and Reporting Tool (BIRT). Before the data can be visualized with BIRT, they have to be processed because BIRT works with the CSV (Comma-separated values) data format. In addition, the profiling results on the host and on the virtual machine are stored in separate files. These files have to be combined into a single input for BIRT. Moreover, the *perf* reports contain both information that is not related to the events and data that are not interested to users. These data have to be moved out.

We developed three scripts to perform the data processing tasks. Their source codes are depicted in Listing 1, Listing 2, and Listing 3 individually. The script RawToCSVs first calls a self-coded program *perf2csv* to transform a *perf* report from its text (ASCII) format to CSV. It then calls the program *splitcsv*, again self-written, to split a CSV file into several individual ones, with each only containing the profiling results of a single event. This makes it easy to produce the required data for visualizing the behavior of a counter event. In case that the user has not used it correctly the script gives an error report.

Listing 1: Source code of the script RawToCSVs.

```
#! /bin/bash
if [ $# -eq 1 ] then
  if [ -f "$1" ] then
    perf2csv $1
    if [ $? -eq 0 ] then
      echo "The report was successfully converted."
      splitcsv $1.csv
      lastindex = $?
      if [ $? -eq -1 ] then
        echo "The report CSV cannot be split"
      else
        echo "The report CSV has been split by Index
              $lastindex."
      endif
    else echo "The report cannot be converted to CSV."
    endif
  else echo "The file does not exist."
  endif
else echo "This program requires a perf report file as
          parameter."
endif
```

For each event there are two CSV files, one for the host and the other for the virtual machines. The script CSVsToCSV combines the two files into a single one. As depicted in Listing 2, it first performs some sorting tasks with the two original CSV files and then calls the *primarykeyjoin* program to put the two files together with the

corresponding routines stored on the same line. The *primarykeyjoin* program is specifically written for this purpose. In the next step, it reduces the amount of lines in the combined CSV file to 10 entries for a better presentation of the event later in the graphical view. Actually, users are also only interested in the routines with a high value of an event.

The last script, as shown in Listing 3, is used to generate diagrams for demonstrating the profiling results in graphical views. The results of the same event for an application on both the host and the virtual machine are contained in a single diagram. The script takes the combined file, generated by CSVsToCSV in the previous step, as input and creates from this file a temporary file for BIRT. The BIRT file contains all information for creating a diagram, including the data, the caption, and the labels for the bars in the diagram. The resulted graphical files are in the form of vectorized SVG (Scalable Vector Graphics) and stored in the same directory as the input file of the script. Finally, the SVG file is processed to produce another two graphical files in the form of EPS and PNG separately.

Listing 2: Source code of the script CSVsToCSV.

```
#! /bin/bash
if [ $# -eq 2 ] then
  if [ -f "$1" -a -f "$2" ] then
    old = ${1//.?.csv/}
    new = ${2//.?.csv/}
    for file in $old.?.csv do
    cat $file | sed -e '1d' | sort -s -t, -k 3,3 |
      sort -s -t, -k 4,4 | sort -s -t, -k 6,6 >
          $file.sorted
    primarykeyjoin $file.sorted
    sort -s -t, -k 3,3 < $file.sorted.key > $file.sorted
    rm $file.sorted.key
    tmp1 = $(dirname $new)
    tmp2 = $(basename $file)
    cat $tmp1/$tmp2 | sed -e '1d' | sort -s -t, -k 3,3 |
      sort -s -t, -k 4,4 | sort -s -t, -k 6,6 >
          $tmp1/$tmp2.sorted
    primarykeyjoin $tmp1/$tmp2.sorted
    sort -s -t, -k 3,3 < $tmp1/$tmp2.sorted.key >
         $tmp1/$tmp2.sorted
    rm $tmp1/$tmp2.sorted.key
    join --check-orde -t, -j 3 $file.sorted $tmp1/$tmp2.
        sorted | sort -s -t, -k 3,3 -n | tail -n 10 >
        $file.combined
    rm $file.sorted
    rm $tmp1/$tmp2.sorted
    done
  else echo "One of the two files is not available."
  endif
else echo "This program needs a split CVS report and a"
     echo "second path to the first file of the report"
     echo "for comparison with the first one."
endif
```

Listing 3: Source code of the script BirtChart.

```
#! /bin/bash
actualdir = $PWD
scriptdir = $(dirname "$(readlink -e "$0") ")
givenfile = $(readlink -e "$1")
if [ $# -eq 4 ] then
  if [ -f "$1" ] then
    if [ -f "$1".svg ] then
       echo "The target file already exists."
    else
       cd "$scriptdir"
       cp birt-chart.rptdesign tmp.rptdesign
       sed -e "s#YYYYYYZZ#$givenfile#g" -i tmp.rptdesign
       sed -e "s#ABCDEFGH#$2#g" -i tmp.rptdesign
       sed -e "s#IJKLMNOP#$3#g" -i tmp.rptdesign
       sed -e "s#QRSTUVWX#$4#g" -i tmp.rptdesign
       export BIRT_HOME=/home/project/bin/birt-run
              time-4_2_0; /home/project/bin/birt-
              runtime-4_2_0/ReportEngine/genReport.
              tmp.rptdesign
       cp image/customl.svg $givenfile.svg
       rm image/customl.svg
       rmdir image
       rm tmp.rptdesign
       rm tmp.html
       cd "$actualdir$"
       inkscape -z -t -e $givenfile.svg.eps
                $givenfile.svg
       inkscape -z -t -e $givenfile.svg.png
                $givenfile.svg
    endif
  else
    echo "The files are not available."
  endif
else
  echo "This program requires an input file with four"
  echo "parameters. 1: the file name with an absolute;
  echo "path; 2: caption of the diagram; 3: name of"
  echo "the blue bar; 4: name of the red bar."
endif
```

In summary, we implemented a set of tools to achieve the goal of showing the applications runtime performance in graphical views with a comparative presentation of the behavior on the host and the virtual machine. The entire working process of the toolchain is demonstrated in Figure 2. It starts with the *perf* tool using its runtime mode of *kvm*. The applications are executed once on the host machine and once on a virtual machine. During the execution of the application *perf* produces a profiling report in the form of a binary file. Using its "report" tool, the binary file is transformed to a text report. The work of our self-developed tools starts with the text report. First, RawToCSVs converts the text report to a CSV form and splits the CSV file into several files with each for a single event. In the following, CSVsToCSV combines the two CSVs with the same event, one for the host and the other for the virtual machine, and reduces the number of items in the combined file in order to highlight the interesting behavior later in the visualization. Finally, BirtChart calls BIRT to creates graphical presentations in three different image formats.

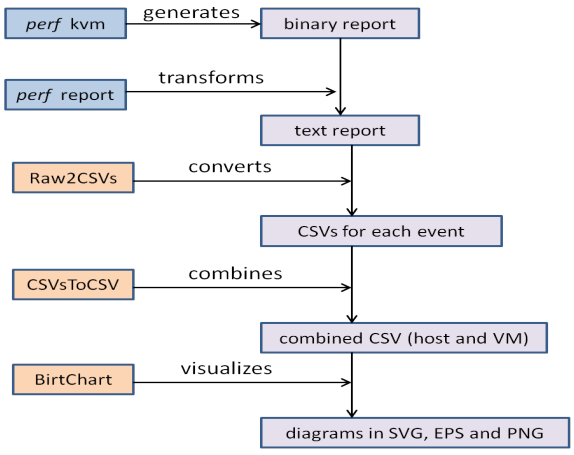

Figure 2: The data flow and functionality of the developed toolchain.

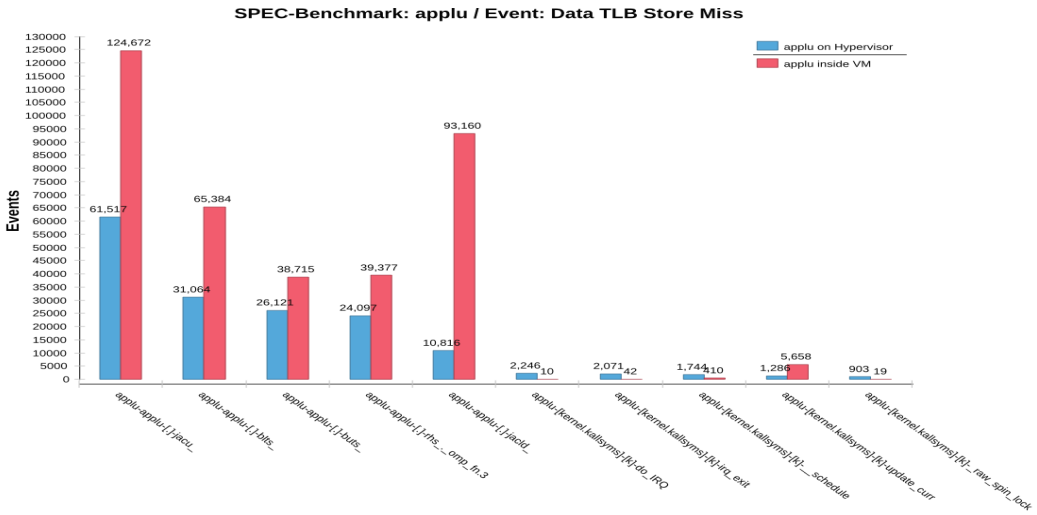

Figure 3: Data TLB Store Miss with the applu application.

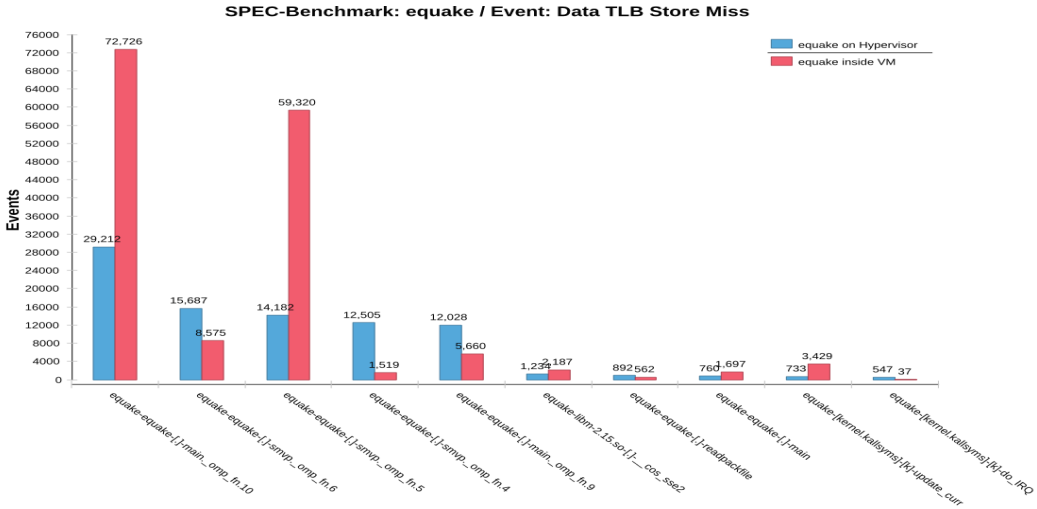

Figure 4: Data TLB Store Miss with the equake application.

SAMPLE RESULTS

We validated the developed toolchain with the NAS and the SPEC parallel benchmarks. We tested a set of applications from both suites. The physical machine used in the testing is an eight-core multiprocessor system equipped with two AMD processor 2356. The total size of RAM is 16GB. The virtual machine is configured with eight VCPUs and 2GB of memory. The applications were run either with a single thread or multiple threads. We measured different counter events, for example, cache miss, TLB miss, context switch and the application execution time.

Figure 3 to Figure 7 shows several sample experimental results. The two diagrams in Figure 3 and Figure 4 are the profiling results of event Data TLB Store Miss with the *applu* and the *equake* application in the SPEC benchmark suite. The profiling was perfromed as the application ran individually either directly on the hypervisor (i.e. host) or within a virtual machine. Each diagram shows the first 10 functions, either in the application code (marked with [.]) or the system libraries (marked with [kernel...]), which have a higher value of the monitored event. To each function there are two bars. The left bar corresponds to the execution on the host machine and the other one is for the virtual machine. It can be commonly seen that the virtual machine produces more TLB misses. For some routines, e.g. the *jacld* function of *applu* in the fifth bar-pair of the upper diagram, our toolchain reports a nine folds data TLB store miss on the virtual machine in contrast to the physical machine.

However, the behavior of this event changes when different applications run simultaneously on the target ma-

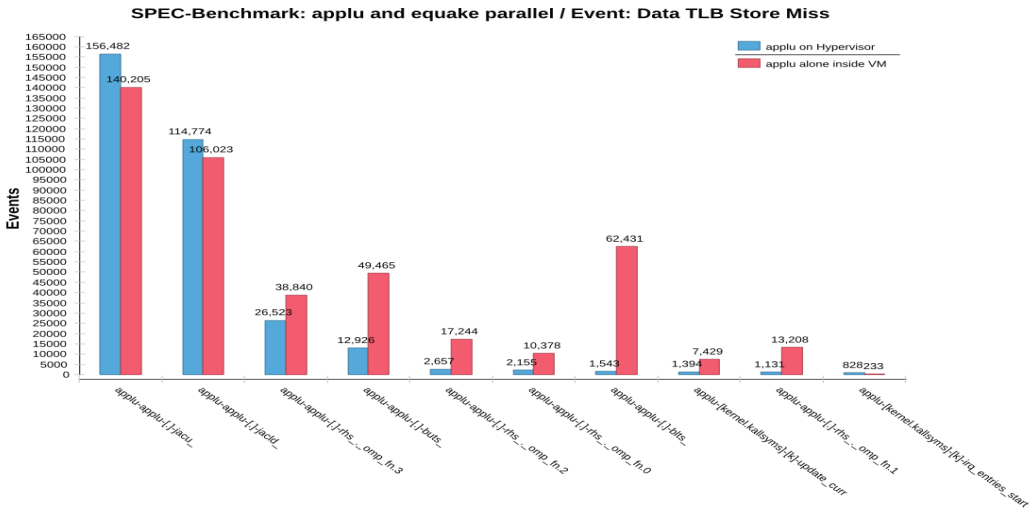

Figure 5: Data TLB Store Miss with parallel run of different applications: results of applu.

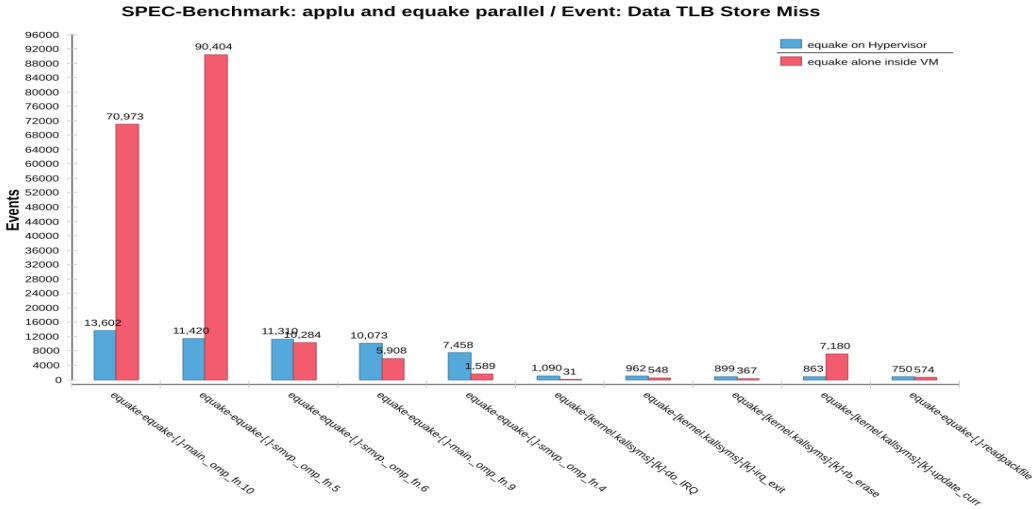

Figure 6: Data TLB Store Miss with parallel run of different applications: results of equake.

chine. Figure 5 and Figure 6 shows the result with the two SPEC applications. For the experiments we first ran both applications on the host in parallel and then ran them separately with each on a single virtual machine. We use again a bar-pair to present the event value with the left bar for running on the host and the right one for running on a single virtual machine. Observing the left two bar-pairs of *applu* in Figure 5, it can be seen that the virtual machine performs better with this event. Comparing the diagram with the one in Figure 3, the reason is clear. The reason is that the host machine produces double of the TLB misses with *applu* when it runs in parallel with *equake*, while the number of the TLB miss produced by the virtual machine remains similarly for the two scenarios. This means that *applu* profits from the parallel run in terms of data TLB miss. Nevertheless, the parallel run enlarged the number of TLB miss of several functions of the *equake* application on the virtual machines, for example, the second bar-pair of the right diagram. This also indicates that there is a large space for programmers to combine the applications for performance tuning.

Besides graphical views in the form of bar charts, our toolchain also provides views of other forms. Figure **??** demonstrates a circle presentation of the counter event "Context Switch". The data was collected by running the BT application from the NAS benchmark suite with two threads. The upper diagram is the result on the host and the lower one is on a virtual machine. Each function is presented with a colored area. The size of the area is related to the number of measured context switches with the corresponding function. A graphical view in such a form can hence highlight the hot functions that have a

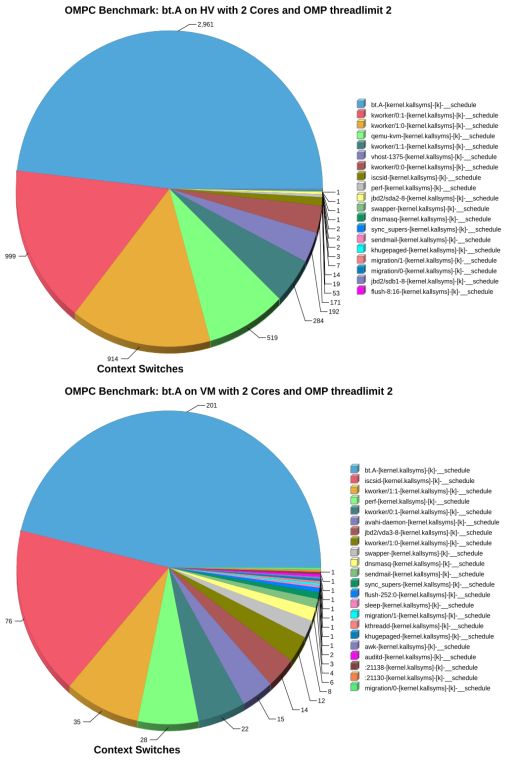

Figure 7: Behavior of event Context Switch with the BT application (upper: host machine, lower: virtual machine).

distinct behavior with a counter event.

CONCLUSIONS

This paper describes a toolchain that was developed to compare the runtime performance of applications on a virtual machine with the behavior on the physical machine. The toolset adopts a low-level Linux profiling tool to collect the raw performance data and then works with the data step by step towards a graphical presentation of the runtime behavior. The paper describes the functionality of the developed tools and shows some experimental results.

The goal of this work is to provide support for programmers to analyze their applications. Hence, for the initial phase we did not perform any optimization with the detected behavior. This can be a future work. Additionally, we plan to study realistic applications with the implemented toolchain.

REFERENCES

Barham, P., Dragovic, B., and Fraser, K. (2003). Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–144.

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

Brunst, H., Hackenberg, D., Juckeland, G., and Rohling, H. (2009). Comprehensive Performance Tracking with Vampir 7. In Mller, M., Resch, M., Schulz, A., and Nagel, W., editors, *Tools for High Performance Computing*, pages 17–29. Springer.

Ciorba, F. M., Groh, S., and Horstemeyer, M. F. (2010). Parallelizing discrete dislocation dynamics simulations on multicore systems. *Procedia CS, International Conference on Computational Science (ICCS 2010)*, 1(1):2135–2143.

de Melo, A. C. (2010). The New Linux 'perf' Tools. In *Proceedings of 17 International Linux System Technology Conference*.

Eclipse. BIRT Project - Business Intelligence and Reporting Tools. http://www.eclipse.org/birt/.

Tikotekar, A. T., et al. (2008). An Analysis of HPC Benchmarks in Virtual Machine Environments. In *Proceedings of Euro-Par 2008 Workshops - Parallel Processing*, volume LNCS 5415, pages 63–71.

Hammond, J. R., Krishnamoorthy, S., Shende, S., Romero, N. A., and Malony, A. D. (2012). Performance Characterization of Global Address Space Applications: A Case Study with NWChem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154.

Intel. Intel VTune Amplifier XE 2013: Performance and Thread Profiler. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

KVM. Kernel Based Virtual Machine. http://www.linux-kvm.org/.

Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., and Lamb, C. (2011). Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proceedings of International Conference on Parallel Processing*, pages 176–185.

Menon, A., Santos, J. R., Turner, Y., Janakiraman, G., and Zwaenepoel, W. (2005). Diagnosing performance overheads in the xen virtual machine environment. In *The 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23.

OProfile. A System Profiler for Linux. http://oprofile.sourceforge.net/.

Shende, S. and Malony, A. D. (2006). The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311.

Sourceforge. perfmon2 - Improving Performance Monitoring on Linux. http://perfmon2.sourceforge.net/.

Youseff, L., Seymour, K., You, H., Dongarra, J., and Wolski, R. (2008). The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 141–152.