2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Characterizing and Reducing Cross-Platform Performance Variability Using OS-level Virtualization

Ivo Jimenez and Carlos Maltzahn
UC Santa Cruz
{ivo,carlosm}@cs.ucsc.edu

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Adam Moody and Kathryn Mohror
Lawrence Livermore National Laboratory
{moody20,kathryn}@llnl.gov

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau
University of Wisconsin-Madison
{remzi,dusseau}@cs.wisc.edu

***Abstract*—Independent validation of experimental results in the field of parallel and distributed systems research is a challenging task, mainly due to changes and differences in software and hardware in computational environments. In particular, when an experiment runs on different hardware than the one where it originally executed, predicting the differences in results is difficult. In this paper, we introduce an architecture-independent method for characterizing the performance of a machine by obtaining a profile (a vector of microbenchark results) that we use to quantify the variability between two hardware platforms. We propose the use of isolation features that OS-level virtualization offers to reduce the variability observed when validating application performance across multiple machines. Our results show that, using our variability characterization methodology, we can correctly predict the variability bounds of CPU-intensive applications, as well as reduce it by up to 2.8x if we make use of CPU bandwidth limitations, depending on the opcode mix of an application, as well as generational and architectural differences between two hardware platforms.**

I. Introduction

A key component of the scientific method is the ability to revisit and reproduce previous experiments. Reproducibility also plays a major role in education since a student can learn by looking at provenance information, re-evaluate the questions that the original experiment answered and thus "stand on the shoulder of giants". In the wide field of computer systems research, the issues of reproducibility manifest when we validate system performance. In order to validate a claim, we need to show that the system performs as stated, possibly in a variety of different scenarios.

When evaluating system performance, multiple variables need to be accounted for; among them are source code changes, compilation and application configuration, as well as workload properties and hardware characteristics. A rule of thumb while executing experiments is: across multiple runs, modify only one variable at a time so that correlation can be accurately attributed to the right variable. Generally, properly enumerating all the environment variables in the software stack is an arduous endeavour; adding hardware to the mix makes it all but impractical since predicting the differences in results that originate when we vary the hardware "variable" is a challenging task. Ideally, given two particular hardware setups, we would like to have a quantifiable expectation of the performance variability for *any* application running on these machines. That is, if we execute an application on one machine and then execute it on another one, we would like to bound the variability (i.e. the performance range) that *any* application running on the former would observe when executed on the latter.

In this work, we initially focus on single-node performance variability, since it is the fundamental building block in distributed and parallel settings. We introduce an architecture-independent method for characterizing the performance of a machine by obtaining a profile (a vector of microbenchark results) that we use to quantify the variability between two hardware platforms. We show this can correctly predict the variability bounds of CPU-intensive applications. We also investigate OS-level virtualization as a way of reducing the expected variability over executions of applications on distinct platforms. OS-level virtualization offers several features for reproducing system performance. While core affinity and memory/swap size limitations have been shown to bring stability across executions in one single system [1], the use of CPU bandwidth limitations can reduce the variability for executions across different platforms. Our experiments show that using CPU bandwidth limitation reduces performance variability by up to 2.8x, depending on the opcode mix of an application, as well as generational and architectural differences between two hardware platforms.

II. OS-level Virtualization

OS-level virtualization is a method where the kernel of an OS allows for multiple isolated user spaces, instead of just one. Such instances (often called containers, virtual private servers (VPS), or jails) may look and feel like a real server from a user's point of view. In the remaining of this paper we focus on Docker [2], which employs Linux's cgroups and namespaces features to provide OS-level virtualization. While

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.97

our discussion is centered around cgroups, the overall strategies can be applied to any of the others.

A. CPU Bandwidth Throttling

Linux's `cgroups` is a unified interface to the operating system's resource management options, allowing users to specify how the kernel should limit, account and isolate usage of CPU, memory, disk I/O and network for a collection of processes. In our case, we're interested in the CPU bandwidth limiting capabilities. `cgroups` exposes parameters for the Completely Fair Scheduler (CFS). The allocation of CPU for a group can be given in relative (`shares`) or absolute (`period` and `quota`) values. Figure 1[1] shows the effect that multiple values for `quota` have on the execution of a CPU-bound process.

B. Limitations of Throttling

While absolute CPU bandwidth limitations work well to isolate processes within a single system, they are not guaranteed to be as effective to reproduce performance across multiple platforms and multiple applications, i.e. finding a value of `quota` on a target machine that would reproduce the performance observed in a base one. The main reason being that fundamental differences between two machines (e.g. CPU and memory bandwidth) make it practically impossible to find a single value for `quota` that works for *every* application. One can find quota/period values that reproduce results for a particular application, but these values won't work for another application with a different opcode mix.

III. Characterizing Performance Variability

Quantifying performance variability across hardware platforms entails characterizing the performance of single machines. While the hardware and software specification can serve to describe the attributes of a machine, the real performance characteristics can only feasibly[2] be obtained by executing programs and capturing metrics at runtime. So the question boils down to which programs should we use to characterize performance? Ideally, we would like to have many programs that execute every possible opcode mix so that we measure their performance. Since this is an impractical solution, an alternative is to create synthetic microbenchmarks that get as close as possible to exercising all the available features of a system.

`stress-ng` is a tool that is used to "stress test a computer system in various selectable ways. It was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces". There are multiple stressors for CPU, CPU cache, memory, OS, network and filesystem. Since we focus on CPU bandwidth, we look at

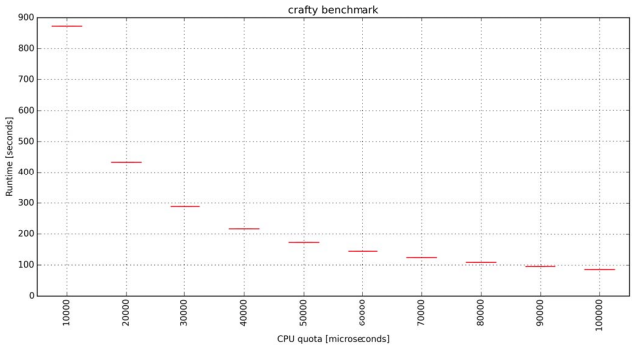

Fig. 1. [source] Boxplots of runtimes of the `crafty` benchmark for multiple values of cpu quota (with a fixed period of 100 microseconds) illustrating the effect of limiting CPU access for a single-threaded process. Every boxplot summarizes 10 executions (interquartile box is tight and overlaps with the median).

the CPU "stressor", which is a routine that loops a function (termed CPU *method*) multiple times and reports the rate of iterations executed for a determined period of time (referred to as `bogo-ops-per-second`). As of version 0.05.09, there are 68 CPU methods that range from bitwise, control flow and floating/integer operations on multiple word sizes.

Using this battery of CPU stressors, one can obtain a profile of CPU performance for a machine. When this profile is normalized against the profile of another machine, we obtain a *variability profile* that characterizes the speedups/slowdowns of a machine B with respect to another one A. We refer to this profile as the variability profile for B/A (or the B/A profile for brevity). Figure 2 shows a histogram (in green) of the variability profile of two machines for all the CPU stressors of `stress-ng`. The purple histogram is discussed in *Section V*.

Given an application, at the hardware level, variability can originate from mainly two sources: hardware generation and major architecture. While it is possible to analyze the performance variation that presents when going from new (faster) to old (slower) architectures, in this work we focus on the opposite (old to new). We believe this is a reasonable assumption since this is the nature of technology advances and mimics the scenario that researchers go through while attempting to reproduce published experiments. Also, we only look at x86_64 and, as shown in *Section V*, we vary between AMD and Intel implementations of this ISA.

IV. Reducing Performance Variability

We now present a methodology that leverages the CPU bandwidth limitation feature of OS-level virtualization to reduce the variability range of performance across distinct hardware platforms. When reproducing performance of an application on a target machine T that originally ran on a base machine B, we propose the following calibration methodology:

1. Execute microbenchmarks on B that characterize the underlying hardware platform.
2. Manipulate absolute cgroup values for the CFS on target machine T in such a way that the performance of

[1]Throughout this article, we include a `source` URL for each figure that links to a github page corresponding to the source code of the experiment that generated this graph.

[2]One can get real performance characteristics by interposing a hardware emulation layer and deterministically associate performance characteristics to each instruction based on specific hardware specs. While possible, this is impractical.

microbenchmarks is as close as possible to the results from machine B; a configuration C_t is obtained.

3. Apply the configuration C_t on machine T and execute the application, which should observe reduced performance variability when compared against the unconstrained execution on T.

If the original execution of the application on base machine B was itself being constrained, then this configuration C_b should be applied in step 1. While we believe this methodology applies to many scenarios, we currently have tested it on single-threaded and non-collocated workloads (see *Results* section).

A. Tuning The Target Machine

Finding the values of CPU bandwidth (step 2) is done via program auto-tunning for one or more microbenchmarks that characterize the performance of the underlying hardware. At every execution step, a docker container is instantiated and constrained with a value for CPU quota. It is reasonable to assume that the performance of CPU with respect to quota allocations resembles a monotonically decreasing function as shown in Figure 1, thus, we can select a random value within the valid tunable range (or alternatively the highest) and climb/descend until we get to the desired performance for the microbenchmark(s) on the target machine. When multiple microbenchmarks are executed their results need to be aggregated (e.g. by taking a weighted average of a speedup metric).

The tuning methodology assumes that the machine where an application is being ported to is relatively more powerful that the one were an application originally ran. When this assumption does not hold, one can resort to constraining the original execution (i.e. generating a C_b for B).

V. Results

In this section we show the effectiveness of our proposed methodology (*Section V.A*) by obtaining the variability profile for a target machine with respect to a baseline; we do so by visualizing the reduction of the variability range when we apply the mapping methodology (*Section IV*) to the target. We then study the effects of this reduction by executing a variety of benchmarks on the same platforms (*Section V.B*). Due to space constraints we omit the detailed description of our experimental setup[3]. We have one target machine T (2012 Xeon E5-2630) whose performance is being characterized with respect to a base machine B (2006 Xeon E5-310). The reason for selecting a relatively old machine as our baseline is two-folded. First, by picking an old machine we ensure that the target machines can outperform the base machine in every test of `stress-ng`. Secondly, having an old computer as part of the our study resembles the scenario that many researchers face while trying to reproduce results found in the literature.

[3]For a complete description please refer to the repository of this article at https://github.com/ivotron/varsys16.

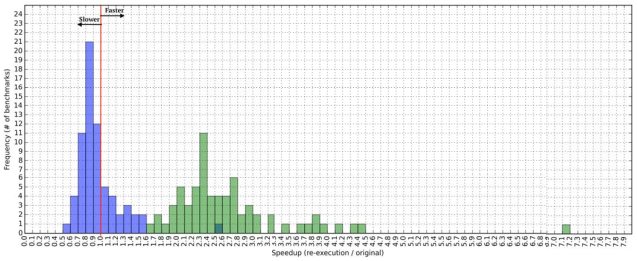

Fig. 2. [source] Histograms for two variability profiles. Each measurement in a histogram corresponds to the performance speedup/slowdown of a `stress-ng` CPU method that a machine has with respect to another one. For example, in the T/B histogram (green), the architectural improvements of machine T cause 11 stressors to have a speedup within the `(2.3, 2.4]` range over machine B.

A. Reduction of Variability Range

Comparing the range of two histograms illustrates the differences in performance variability for a pair of machines. Perfect performance reproducibility of results would result in having the performance of every benchmark to be in $x = 1.0$. As mentioned before (*Section II.B*), fundamental differences between two machines such as CPU, memory, micro-controllers and BIOS configuration make it practically impossible to have perfect reproducibility between two platforms.

Yet, reducing the performance variability (shrinking the range around $x = 1.0$) is an attainable goal. The green histogram in Figure 2 corresponds to variability profile T/B. The purple one corresponds to the variability profile of T after being constrained using the tuning methodology from *Section IV*. We denote this profile as T'/B. In this particular case, tuning resulted in a CPU quota of 6372 microseconds for a period of 10000 microseconds. When these bandwidth limitations are in place, the variability range is reduced from $[1.65, 7.10]$ to $[0.60, 2.54]$, i.e. from a range of length 5.45 to one of size 1.94, a ~2.8x reduction.

We make two main observations about Figure 2. First, more than 50% of the data points cluster around the $[0.78, .98]$ range (with 88 as the median), while ~25% around the $[0.83, 0.93]$ range (not shown) for the limited case (purple histogram). In the unconstrained case (green), the median is 2.48 (mean is 2.70), with a long tail towards the higher speedup values. Secondly, while 6372 represents ~63% of CPU time, the range shrinks only by ~50%. As shown in Figure 1, this is mainly due to the non-linear behavior of CPU performance under different loads. An open question is whether the same performance variability would be observed at the hardware level by using dynamic frequency scaling.

B. Validation of Variability Characterization

Assuming `stress-ng`'s distinct CPU methods represent a realistic coverage of the multiple physical features of a processor, we can reasonably assume that the performance of applications with and without constrained CPU bandwidth will land within the range obtained by our variability characterization profiles introduced in *Section III*. In order to

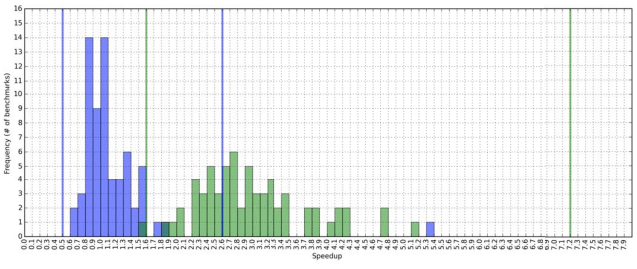

Fig. 3. [source] Histogram for T/B and T'/B profiles. Measurements come from the following benchmarks: `STREAM`, `cloverleaf-serial`, `comd-serial`, `sequoia (amgmk, crystalmk, irsmk)`, `c-ray`, `crafty`, `unixbench`, `stress-ng` (string, matrix, memory and cpu-cache). Vertical lines denote the limits of the predicted variability range (Figure 2), obtained from executing `stress-ng` CPU stressors. Points outside the predicted line correspond to `STREAM`. The rightmost point for the unconstrained (green) histogram is not shown to improve the readability of the figure; it lies on the 14x bin.

corroborate this hypothesis, we executed 66 benchmarks with and without CPU bandwidth limitations on the target system. Every benchmark on this and the previous section was executed on the base and target systems in docker containers. In order to minimize the variability that might originate from distinct compiler optimizations we disable compiler optimizations (`gcc`'s `-O0` flag). Also, as mentioned previously, these are single-threaded processes running in uncontended systems; our goal is to generate bounded performance rather than perfect reproducibility (*Section V.B*).

Figure 3 shows the results of our tests for both unconstrained (green; T/B profile) and unconstrained (purple; T'/B profile) scenarios. Each point on a histogram corresponds to one benchmark. The two vertical lines denote the variability range obtained from Figure 2. For the constrained case (purple), with the exception of one point, all executions land within the predicted range. We also observe that while the highest value of the range obtained in the previous section (rightmost vertical purple line) is in the 2.6x bin, the performance of 64 out of 66 never go above the smaller $[0.6-1.6]$ range. In the case of executions without limits (green histogram), we observe 2 points going out of the predicted range, the one at $[1.5-1.6]$ and another (not shown) at 14x, both corresponding to memory-bound benchmarks (`stress-ng-memory-malloc` and `STREAM`, respectively).

From the analysis of the variability profiles for these 66 benchmarks, we can conclude that the set of `stress-ng` microbenchmarks are good representatives of CPU performance and thus they can serve to characterize a machine for CPU-intensive workloads. Also, the variability profile seems to be a good performance predictor, i.e. an execution lies within the determined speedup/slowdown range.

VI. Related Work

The challenging task of evaluating experimental results in applied computer science has been long recognized [3], where the focus is more on numerical reproducibility rather than performance evaluation. In systems research, runtime performance *is* the subject of study, thus we need to look at it as a primary issue.

The closest work to our approach is Fracas [4]. Fracas emulates CPU frequency for the same machine. As reported in [4], accurately emulating CPU frequencies is a challenging task, even in the same system. Instead, we take the performance profiles as our baseline and quantify variability, irrespective of the differences between frequencies.

Architecture-independent characterization of workloads [5] and performance [6] has been extensively researched in the past. In our case, working at the OS virtualization level imposes new challenges. As we have shown, a way of overcoming these is by using a comprehensive list of microbenchmarks that can accurately characterize the performance of the underlying system.

VII. Conclusion

Characterizing the variability between machines significantly facilitates the interpretation of results when validating performance reproducibility across distinct platforms. While performance models and hardware emulation can, in principle, accurately capture performance characteristics of hardware, it comes at extreme cost and difficulty. In this work we have introduced a simpler model for validating results that relies on performance profiles and incorporates variability ranges. With the aid of OS-level virtualization we can reduce the variability by limiting CPU bandwidth.

Acknowledgements: Work performed under auspices of US DOE by LLNL contract DE-AC52-07NA27344 CONF-681457 and by SNL contract DE-AC04-94AL85000 .

VIII. References

[1] D. Beyer, S. Löwe, and P. Wendler, "Benchmarking and Resource Measurement," *Model Checking Software*, 2015.

[2] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, Mar. 2014.

[3] J.P. Ignizio, "On the Establishment of Standards for Comparing Algorithm Performance," *Interfaces*, vol. 2, Nov. 1971.

[4] T. Buchert, L. Nussbaum, and J. Gustedt, "Accurate Emulation of CPU Performance," *Euro-Par 2010 Parallel Processing Workshops*, 2010.

[5] K. Hoste and L. Eeckhout, "Microarchitecture-Independent Workload Characterization," *IEEE Micro*, vol. 27, May. 2007.

[6] G. Marin and J. Mellor-Crummey, "Cross-architecture Performance Predictions for Scientific Applications Using Parameterized Models," *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.