Accurate emulation of CPU performance

Tomasz Buchert[1], Lucas Nussbaum[2], and Jens Gustedt[1]

[1] INRIA Nancy – Grand Est
[2] LORIA / Nancy-Université

Abstract. This paper addresses the question of CPU performance emulation, which allows experimenters to evaluate applications under a wide range of reproducible experimental conditions. Specifically, we propose Fracas, a CPU emulator that leverages the Linux Completely Fair Scheduler to achieve performance emulation of homogeneous or heterogeneous multi-core systems. Several benchmarks reproducing different types of workload (CPU-bound, IO-bound) are then used to thoroughly compare Fracas with another CPU emulator and hardware frequency scaling. We show that the design of Fracas results in a more accurate and a less intrusive CPU emulation solution.

1 Introduction

The evaluation of algorithms and applications for large-scale heterogeneous platforms is a very challenging task. Different approaches are in widespread use [5]: *simulation* of course, but also *in-situ* experiments (where a real application is tested on a real environment), and *emulation* (where a real application is tested on a simulated environment).

It is often difficult to perform experiments in a real environment that suits the experimenter's needs: the available infrastructure might not be large enough or have the required characteristics. Moreover, controlling experimental conditions in heterogeneous and distributed systems, like grids or the Internet, makes the experimental validation error-prone. Therefore, *in-situ* experiments are often not feasible, and the use of an emulated or simulated environment is often preferred. Many distributed system emulators (e.g. MicroGrid [9], Modelnet [11], Emulab [12], Wrekavoc [2]) have been developed over the years, but most of them focus on network emulation.

Surprisingly, the question of the emulation of CPU speed and performance is rarely addressed by them. However, it is crucial to evaluate applications under a set of different experimental conditions: to know how application's performance is related to the performance of the CPU (as opposed to the communication network), or how an application would perform when executed on clusters of heterogeneous machines, with different CPUs.

This paper explores the emulation of CPU performance characteristics, and proposes a new implementation of a CPU emulator: Fracas. After exposing the related works in Section 2, the problem is clarified and formalized in Section 3. Fracas is then described in Section 4, and evaluated extensively in Section 5.

2 Related Work

Several technologies and techniques enable the execution of applications under a different perceived CPU speed.

Dynamic frequency scaling (known as *Intel SpeedStep*, *AMD PowerNow!* on laptops and *AMD Cool'n'Quiet* on desktops and servers) is a hardware technique to adjust the frequency of CPUs, mainly for power-saving purposes. The frequency may be changed automatically by the operating system according to the current system load, or set manually by the user. In most CPUs, those technologies only provide few frequency levels (around 5), but some CPUs provide a lot more (11 levels on *Xeon X5570*, ranging from 1.6 GHz to 2.93 GHz).

Frequency scaling has the advantage of not causing overhead, since it is done in hardware. It is also completely *transparent*: applications cannot determine whether they are running under CPU speed degradation unless they read the operating system settings. On the other hand, the main drawback of *frequency scaling* is the small number of scaling levels available, which might not be sufficient for some experiments. It should also be noted that it is usually not possible to change the frequency of the processor cores independently, and that dynamic frequency scaling indirectly affects memory access speed as will be shown in Section 5.

CPU-Lim is the CPU limiter implemented in Wrekavoc [2]. It is implemented completely in user-space, using a real-time process that monitors the CPU usage of programs executed by a predefined user. If a program has a too big share of CPU time, it is stopped using the SIGSTOP signal. If, after some time, this share falls below the specified threshold, then the process is resumed using the SIGCONT signal. The measure of CPU load of a given process is approximated by:

$$\text{CPU usage} = \frac{\text{CPU time of the process}}{\text{current time} - \text{process creation time}}$$

CPU-Lim has the advantages of being simple and portable to most POSIX systems (the only non-conformance is the reliance on the `/proc` filesystem). However, it has several drawbacks.

Poor scalability: CPU-Lim polls the `/proc` filesystem with a high frequency to measure CPU usage and to detect new processes created by the user. This introduces a high overhead in the case of a large number of running processes. The polling interval also needs to be experimentally calibrated, as it influences the results of the experiments.

Not transparent: A malicious program can detect the effects of the CPU degradation and interfere with it by blocking the SIGCONT signal or by sending SIGCONT to other processes.

Incorrect measurement of CPU usage: The CPU usage is computed *locally* and independently for every process. If four CPU-bound processes in the system consisting of one core are supposed to get only 50% of its nominal CPU speed, then every process will get 25% of the CPU time. Every process has its CPU

usage below a specified threshold, yet the total CPU usage is 100%, instead of the expected 50%. Additionally, the method gives sleeping processes an unfair advantage over CPU-bound processes because it does not make any distinction between sleeping time (e.g. waiting for IO operation to finish) and time during which the process was deprived of the CPU.

Multithreading issues: CPU-Lim works at the *process* level instead of the *thread* level: it completely ignores cases where multiple threads might be running inside a single process for its CPU usage computation. Therefore, one may expect problems in degrading CPU speed for multithreaded programs.

KRASH [8] is a CPU load injection tool. It is capable of recording and generating reproducible system load on computing nodes. While it is not a CPU speed degradation method *per se*, similar ideas have been used to design Fracas, which is presented later in this paper.

Using special features and properties of the Linux kernel to manage groups of processes (*cpusets*, *cgroups*), a CPU-bound process is created on every CPU core and assigned a desired portion of CPU time by setting its available CPU share. The system scheduler (*Completely Fair Scheduler*) then distributes the CPU time at the *cpuset* level and later in each *cpuset* independently, resulting in the desired CPU load being generated for each core.

Although this method relies on several recent Linux-specific features and interfaces and is not portable to different operating systems, however it has several advantages. First, it is completely transparent, since it works at the kernel level. Processes cannot notice the injected load directly nor interfere with it. Second, this approach is very scalable with the number of controlled processes: no polling is involved, and there are no parameters to calibrate. There are, however, a few settings of the Linux scheduler that affect the accuracy of the result, as discussed later (see Section 4).

While there are many virtualization technologies available, due to their focus on performance none of them offer any way to emulate lower CPU speed: they only allow to restrict a virtual machine to a subset of CPU cores, which is not sufficient for our purposes. It is also possible to take an opposite approach, and modify the virtual machine hypervisor to change its perception of time, giving it the impression that the underlying hardware runs faster or slower [4].

Another approach is to emulate the whole computer architecture using the Bochs Emulator [1], which can be configured to perform a specific number of "emulating instructions per second". However, according to Bochs's documentation, that measure depends on the hosting operating system, the compiler configuration and the processor speed. As Bochs is a fully emulated environment, this approach introduces performance impact that is too high for our needs. Therefore, it is not covered in this paper.

3 Problem Statement

In this section *core* is the smallest processing unit that can execute the code of the program independently on a processor. It is equivalent to a core of a physical

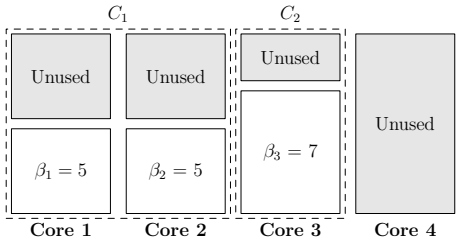

Fig. 1: An example of a CPU emulation problem. Here: $N = 4$, $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 10$, $M = 2$, $C_1 = \{1, 2\}$, $C_2 = \{3\}$, $\beta_1 = \beta_2 = 5$, $\beta_3 = 7$, $\beta_4 = 0$.

processor. Consequently, *processor* is a set of cores and is equivalent to a physical processor. Additionally, a distinction is made between *real* processor/core (the one existing as a hardware implementation) and *emulated* processor/core (the one being emulated).

Let's assume that a computer system consists of N cores with speeds $\alpha_1 \leq \alpha_2 \leq \alpha_3 \leq \ldots \leq \alpha_N$. The goal is to emulate M processors, using this physical processor. The m-th emulated processor, denoted C_m occupies a subset of real cores: $C_m \subset \{1, 2, \ldots, N\}$. None of the physical cores will be occupied by more than two emulated ones so $C_i \cap C_j = \emptyset$ for $1 \leq i < j \leq M$.

Finally, for each emulated processor C_m $(1 \leq m \leq M)$, a core $k \in C_m$ has the emulated speed β_k. If $k \notin C_m$ for every $1 \leq m \leq M$ then by definition $\beta_k = 0$.

It is also reasonable to assume that $\alpha_i \geq \beta_i$ for $i \in \{1, \ldots, N\}$, so that each emulated core can be mapped to a physical one. Also, in most real-life scenarios it is true that $\alpha_1 = \alpha_2 = \alpha_3 = \ldots = \alpha_N$. If not stated differently, this is always assumed in the following sections.

An example of the problem instance is presented in Figure 1.

The following special cases of this problem are of particular interest and are considered in this paper:

(A) $M = 1$ and C_1 has one element – a single core processor is emulated.
(B) $M = 1$ and C_1 has exactly N elements – the only emulated processor spans all physical cores.

This is a hardly a complete formalization of the general problem. In a more general setting one may relax some previous assumptions or take other properties of the computer systems into account: speed of the random access memory, the CPU cache size and properties, Simultaneous Multi Threading (SMT) (e.g. Intel Hyper-Threading technology) or Non-Uniform Memory Architectures (NUMA).

4 Proposed Solution

Fracas is using an approach similar to KRASH. On every processor core a CPU-intensive process is created. It burns a required amount of CPU cycles on its

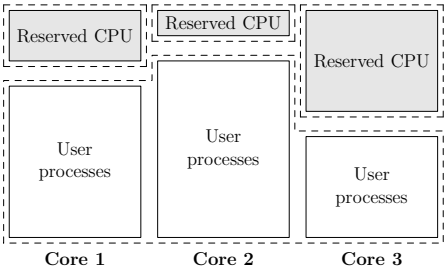

Fig. 2: The idea behind Fracas

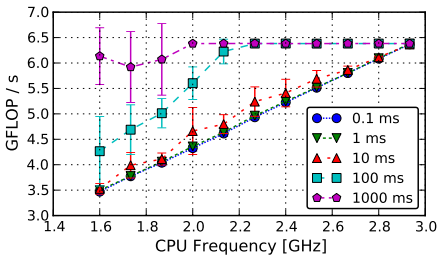

Fig. 3: Latency of the scheduler & Fracas

core. All other tasks in the system are moved to another group which spans all cores. CPU time is distributed to groups proportionally to their weights so, by adjusting them properly, the latter group will acquire the desired amount of the CPU time. Figure 2 presents the idea graphically.

This method uses Completely Fair Scheduler (CFS) by Ingo Molnar which is a default scheduler in the current Linux release (2.6.34). It was merged into kernel mainline in version 2.6.23. Cpusets, which also play an important role, were introduced in version 2.6.12 of the Linux kernel. The O(1) scheduler (also by Ingo Molnar) used back then does not possess the features as required by Fracas [8].

The following CFS parameters [7] have been experimentally verified to have impact on the work of Fracas: `latency` (default kernel value: 5ms) – targeted preemption latency for CPU-bound tasks, and `min_granularity` (default kernel value: 1ms) – minimal preemption granularity for CPU-bound tasks. The first one defines the time which is a maximum period of a task being in a preempted state and the latter is a smallest quantum of CPU time given to the task by the scheduler.

Ignoring rounding, the kernel formula for computing the period in which every running task should be ran once is (n_r - a number of running tasks) $\max(n_r \cdot \texttt{min\_granularity}, \texttt{latency})$. Therefore, setting `latency` and `min_granularity` to the lowest possible values (which is 0.1ms for both of them) will force the scheduler to compute the smallest possible preemption periods and, as a result, the highest possible activity of the scheduler. This substantially improves the accuracy of Fracas (see Figure 3). In this figure each plot presents the result for `Linpack` benchmark (see Section 5.2) under different scheduler latency. As can be seen, the lower the latency, the more the results converge to the perfect behavior.

5 Evaluation

In the following sections three different methods are evaluated which can be used to emulate the CPU speed: dynamic frequency scaling (abbreviated to *CPU-Freq*), CPU-Lim and Fracas.

There are many pitfalls related to the experiments involving processors. Contemporary processors have a very complex architecture – due to cache, branch prediction, simultaneous multithreading technology, code alignment in the memory and other factors, the behavior of programs may vary significantly in similar conditions. Another problem is posed by external factors that may change the execution conditions on the fly. For instance, dynamic frequency scaling is used to conserve power or to generate less heat than during a normal operation. Preferably this feature should be turned off during all the experiments. Nevertheless, even if turned off, most CPUs may also throttle their frequency down in the case of dangerous overheat, leading to an unexpected performance loss. To make things even worse, the newest Intel processors in the Nehalem family (used in our experiments) may introduce an "unexpected" performance gain: *Turbo Mode* technology allows a processor core to overclock itself when the other cores are idle. In the following experiments this technology was knowingly turned off as well as Intel Hyper-Threading.

The experimental scientist must be aware of these problems to perform the experiments reliably.

5.1 Experimental Setup

All experiments were performed on the Grid'5000 experimental testbed [3]. Specifically, the following clusters were used:

- The *Parapide* cluster located in Rennes, France.
  All nodes in the cluster have two quad-core Intel processors (Intel Xeon X5570). Each core has 11 different levels of dynamic frequency scaling available.
- The *Chti* cluster located in Lille, France.
  All nodes in the cluster have a pair of single-core AMD processors (AMD Opteron 252). Finally, this CPU model offers 6 levels of dynamic frequency scaling.

All nodes from a given cluster offer exactly the same configuration so it was possible to perform experiments in parallel. To achieve this, a client-server application was created to distribute the tests automatically. The order in which tests are distributed is randomized. Nodes involved in the experiments were deployed with the same instance of Linux operating system (kernel version: 2.6.33.2).

The experimental framework as well as instructions to reproduce the results are available at `http://www.loria.fr/~lnussbau/files/fracas.html`.

5.2 Benchmarks

The following benchmarks, testing important aspects of the CPU emulation, were used:

- `Linpack` (GFLOP/s) – a well known benchmark used to measure floating point computing power. The version used is a slightly modified version included in the HPCC Benchmark suite (version 1.4.0, released 2010-03-26) [6].

- `Sleep` (Loops/s) – a test performing CPU-intensive work, sleeping for the amount of time that was required to perform the work, and finally running the same computation once again. The result is the number of the computation cycles performed divided by the the time of the whole computation.
- `UDP` (Sends/s) – a program that measures the time required to send many UDP packets to the network. The result is a number of `sendto()` invocations divided by the time required to perform them.
- `Threads` (Loops/s) – a benchmark that creates a few threads (5 threads for the Parapide cluster and 2 threads for the Chti cluster). After a simple integer computation all threads are joined (using `pthread_join`) and the result is the number of computation cycles performed by each thread divided by the time required to join all of them.
- `Processes` (Loops/s) – a modification of `Threads` benchmark. Instead of the threads, processes are created. They are joined using `waitpid` syscall.
- `STREAM` (GB/s) – a synthetic benchmark that measures sustainable memory bandwidth. It is available at [10].

Each benchmark performs a small calibration loop at the beginning to assure that the computation time is big enough as to yield meaningful results (i.e. it's not affected by the granularity of system clock). Please also note that the results from different benchmarks, even though sometimes measured in the same units, are not comparable in any sensible way.

5.3 Results and Discussion

All tests were performed ten times each and the final plot value is the average of all results. The whiskers describe the 95% confidence intervals of this value. The results from the Chti cluster are attached only if they significantly differ from the results obtained on the Parapide cluster. The majority of the results is identical and differences can be easily explained. This further convinces us that the results are independent and general. Most of the time the results obtained by CPU-Freq method are used as a reference, as a model we want to emulate using other methods.

For every emulated frequency f, let's define $\mu = \frac{f}{f_{max}}$ as a *scaling ratio* (where f_{max} is the maximum processor speed).

For a CPU intensive work the execution speed should be proportional to the ratio μ. In Figure 4 one can see that all three methods behave similarly for a CPU intensive work. Nevertheless CPU-Lim gives less predictable results and the slope of a plot with Fracas results is different than the one obtained from CPU-Freq. The observed difference between Fracas and CPU-Freq while emulating processor at 1.6 GHz speed is around 2.5%. This shows that dynamic frequency scaling on Intel processors affects the performance by a different factor than just the ratio μ.

The time when processes sleep, either voluntarily or waiting for IO operation to finish, should not influence the behavior after the process is woken up. However, from Figure 5 it is evident that CPU-Lim has problems with controlling

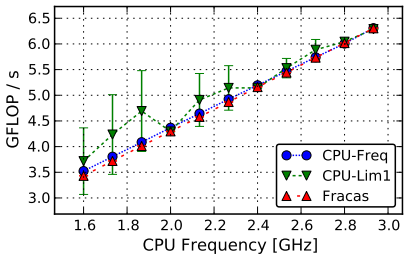

Fig. 4: `Linpack` benchmark

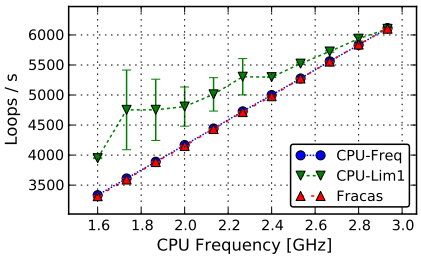

Fig. 5: `Sleep` benchmark

processes which perform this type of work. Both Fracas and CPU-Freq behave as expected.

Generally, IO operations should not be affected by the CPU scaling because they depend on the hardware traits (like network card speed). Results from the Parapide cluster show that the time required to perform intensive access to the hardware does not scale with emulated CPU speed on the tested Intel processor. However, the results from the Chti cluster show (see Figure 9) that it scales by a factor of 16% when emulating the lowest possible frequency using CPU-Freq. It is because the AMD Opteron 252 processor has a wider range of available frequencies than Intel Xeon X5570 (but a smaller set of possible values). If scaled to 1.0 GHz, the time required to prepare UDP packet is becoming a significant factor. This is a proper behavior of all methods.

The CPU time is a resource shared by all the tasks running in the system. All the methods should scale down the total CPU usage and not only the one perceived by every process. Multiple tasks doing the same work simultaneously on different cores should finish at the same time and the total time should be roughly the same as the CPU time consumed by one task. In Figure 6 and Figure 7 the results for this kind of work are presented. A strange behavior of Fracas was observed – the time required to finish the work is much longer than the expected time. This odd behavior is of course a wrong one. CPU-Lim performs much better but its results are very unstable. Additionally, a significant overhead of CPU-Lim method can be observed when used to control even just 5 processes – the results of CPU-Lim method oscillate in the range $77\% \div 89\%$ of the respective CPU-Freq result (excluding the case of emulating the highest possible frequency when CPU-Lim processes are not started at all).

The only significant difference between Figure 7 and Figure 6 is the behavior of CPU-Lim. The observed phenomenon was described in Section 2 as *Incorrect measurement of CPU usage* – the whole process (consisting of 5 threads) is controlled and the CPU usage is an accumulated value from all the threads. Therefore, CPU-Lim stops the process too often. As predicted, each result of CPU-Lim equals almost exactly 20% percent of CPU-Freq's one.

Generally, the memory speed is expected to not change at all while scaling CPU speed down. The conclusion from the data from Figure 8 is that memory

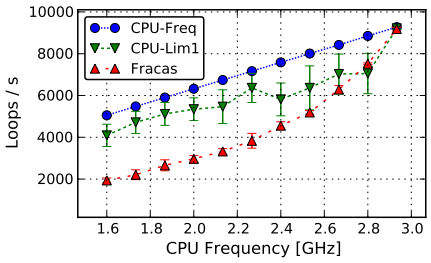

Fig. 6: `Processes` benchmark

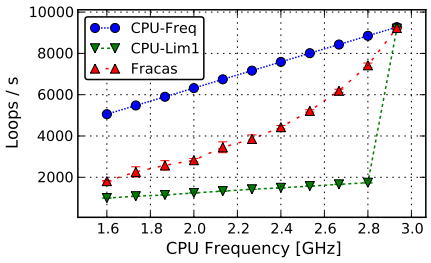

Fig. 7: `Threads` benchmark

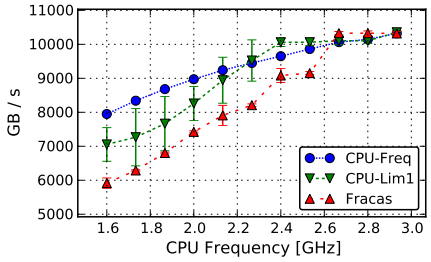

Fig. 8: `STREAM` benchmark

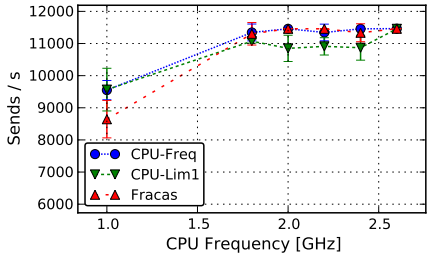

Fig. 9: `UDP` benchmark (on Chti cluster)

speed is indeed affected by every presented method and by each method in its own way. Interestingly dynamic frequency scaling does not change memory speed linearly (as opposed to the pure computation speed, as can be seen in Figure 4).

All the above observations are summarized in a less formal way in Table 1.

Table 1: Summary of the presented emulation methods

	CPU-Freq	CPU-Lim	Fracas
Granularity of emulation	Coarse	Very good	Very good
Accuracy of results	Excellent	Mediocre	Depends on work
Stability of emulation	Excellent	Mediocre	Very good
Scalability (with no. of tasks)	Unlimited	Very bad	Very good
Intrusiveness	None	Very high	Almost none

6 Conclusions

Unfortunately, the obtained results show that none of the presented methods is perfect. Dynamic frequency scaling provides the best results, but its applicability

is very limited due to its coarse granularity of CPU speed emulation, preventing the emulation of arbitrary speeds. Similarly, Fracas is a very good solution for the single thread/process case, and provides notable improvements compared to CPU-Lim, especially regarding accuracy and intrusiveness, but exhibits some problems in the multi-thread/process case.

In our future work, we plan to make further improvements to Fracas. First, we will try to solve the problems shown in the multi-thread/process case. Second, we will try to incorporate the emulation of other CPU characteristics, like memory bandwidth, as it becomes a crucial characteristic of modern CPUs. We would also like to emulate common features such as simultaneous multi-threading. The ultimate goal is to create a reliable, fine-grained solution to cover all important aspects of CPU emulation.

In order to provide an easy way to run experiments with Fracas, we will integrate it into the Wrekavoc emulator, enabling experimenters to combine CPU emulation with Fracas, and network emulation on large clusters.

References

1. Bochs documentation. `http://bochs.sourceforge.net/`
2. Canon, L.C., Dubuisson, O., Gustedt, J., Jeannot, E.: Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool. Journal of Systems and Software 83, 786–802 (2010)
3. The Grid'5000 experimental testbed. `https://www.grid5000.fr`
4. Gupta, D., Vishwanath, K.V., Vahdat, A.: Diecast: testing distributed systems with an accurate scale model. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (2008)
5. Gustedt, J., Jeannot, E., Quinson, M.: Experimental Validation in Large-Scale Systems: a Survey of Methodologies. Parallel Processing Letters 19, 399–418 (2009)
6. HPC Challenge Benchmark. `http://icl.cs.utk.edu/hpcc/`
7. Jones, M.T.: Inside the Linux 2.6 Completely Fair Scheduler: Providing fair access to CPUs since 2.6.23. `http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/`
8. Perarnau, S., Huard, G.: Krash: reproducible CPU load generation on many cores machines. In: IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing (2010)
9. Song, H.J., Liu, X., Jakobsen, D., Bhagwan, R., Zhang, X., Taura, K., Chien, A.: The microgrid: a scientific tool for modeling computational grids. In: Proceedings of IEEE Supercomputing (2000)
10. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream/`
11. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., Becker, D.: Scalability and accuracy in a large-scale network emulator. SIGOPS Oper. Syst. Rev. 36(SI), 271–284 (2002)
12. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: OSDI02. Boston, MA (2002)