Methods for Emulation of Multi-Core CPU Performance

Tomasz Buchert, Lucas Nussbaum, Jens Gustedt

▶ To cite this version:

Methods for Emulation of Multi-Core CPU Performance

Tomasz Buchert, Lucas Nussbaum and Jens Gustedt

INRIA Nancy–Grand Est - LORIA - University of Nancy, France

Email: Firstname.Lastname@inria.fr

Abstract—**When validating or evaluating real distributed applications, it is useful to be able to test the application in a large range of environments. In that context, emulation of CPU performance enables researchers to investigate how the performance of the application is affected by the performance of the participating CPUs. Using a homogeneous cluster of fast multi-core nodes, it is therefore possible to evaluate how an application would behave on a heterogeneous set of nodes, with varying performance and number of cores.**

In this paper, three new methods for the emulation of CPU performance in the multi-core case are proposed: Fracas, CPU-Gov, and CPU-Hogs. Fracas relies on smart configuration of the Linux scheduler to achieve the desired emulation, CPU-Gov leverages the hardware CPU frequency scaling, and CPU-Hogs is a multi-core implementation of a CPU burner. These methods are compared and evaluated together with existing methods, with a set of micro-benchmarks, and show significant improvements over state-of-the-art solutions.

Keywords-**emulation; multi-core; CPU performance; experimental validation**

I. INTRODUCTION

The evaluation of algorithms and applications for large-scale distributed platforms such as grids, cloud computing infrastructures, or peer-to-peer systems is a very challenging task. Different approaches are in widespread use [1]: *simulation* of course (where the target is modeled, and evaluated against a model of the platform), but also *in-situ* experiments (where a real application is tested on a real environment, like PlanetLab or Grid'5000). A third intermediate approach, *emulation*, consists in executing the real application on a platform that can be altered using special software or hardware, to be able to reproduce desired experimental conditions.

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, nor have the required characteristics regarding performance or reliability. Furthermore, modifying the experimental conditions often requires administrative privileges which are rarely given to normal users of experimental platforms. Therefore, *in-situ* experiments are often of relatively limited scope: they tend to lack generalization and provide a single data point restricted to a given platform, and should be repeated on other experimental platforms to provide more insight on the performance of the application.

The use of emulators can alleviate this, by enabling the experimenter to change the performance characteristics of a given platform. Since the same platform can be used for all the experiment, it is easy to conclude on the influence of the parameter that was modified. However, whereas many distributed system emulators (e.g MicroGrid [2], Modelnet [3], Emulab [4], Wrekavoc [5]) have been developed over the years, they mostly focus on network emulation: they provide network links with limited bandwidth or increased latency, complex topologies, etc.

Surprisingly, the question of the emulation of CPU speed and performance is rarely addressed by existing emulators. This question is however crucial when evaluating distributed applications, to know how the application's performance is related to the performance of the CPU (as opposed to the communication network), or how the application would perform when executed on clusters of heterogeneous machines.

This paper explores the emulation of CPU performance characteristics in the context of multi-core systems, and proposes three new methods for CPU emulation. After exposing the related work in Section II, the three methods are described in Section III and evaluated extensively with a set of micro-benchmarks (in Section IV).

II. RELATED WORK

Several technologies and techniques enable the execution of applications under a different perceived or real 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. For example, Linux exposes a frequency scaling interface using its *sysfs* pseudo-filesystem, and provides several *governors* that react differently to changes of system load. In most CPUs, those technologies only provide a few frequency levels (in the order of 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
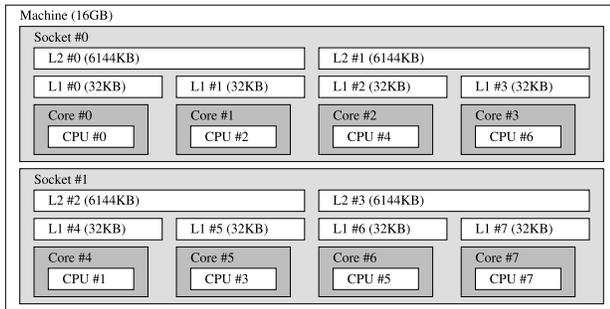
Figure 1: Architecture of a dual *Intel L5420* machine, as shown by `hwloc` [6]. Cores sharing their L2 cache cannot change speed independently.

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. Another drawback is that on most CPUs, it is not possible to change the frequency of each core independently, as exposed by Linux in `cpufreq/related_cpus` (Figure 1). Finally, dynamic frequency scaling indirectly also affects memory access speed as will be shown in Section IV.

CPU-Lim is the CPU limiter implemented in Wrekavoc [5]. It is implemented completely in userspace, 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 [7] is a CPU load injection tool. It is capable of recording and generating reproducible system load on computing nodes. It is not a CPU speed degradation method *per se*, but similar ideas have been used to design one of the methods presented later in this paper, i.e., Fracas.

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.

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 latency of the scheduler, and thus the accuracy of the result, as discussed in [8].

Although 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 (*time dilation*), giving it the impression that the underlying hardware runs faster or slower [9].

Another approach is to emulate the whole computer architecture using the Bochs Emulator, 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
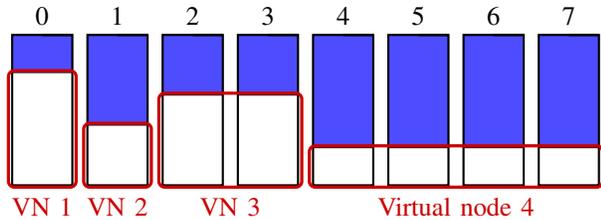
Figure 2: Multi-core CPU emulation using a 8-core machine decomposed into 4 virtual nodes, using respectively 1, 1, 2 and 4 cores, allocated respectively 75%, 40%, 60% and 25% of the physical cores' performance.
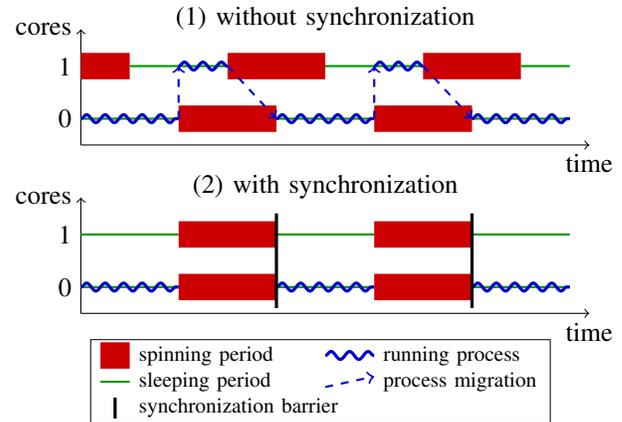

Figure 3: CPU-Hogs: using CPU burners to degrade CPU performance. Without synchronization between the spinning threads, the user process will migrate between cores and use more CPU time than allocated. This is solved in CPU-Hogs by using a synchronization barrier: there is then no advantage for the user process to migrate between cores.

performance impact that is too high for our needs. Therefore, it is not covered in this paper.

III. METHODS FOR MULTI-CORE CPU EMULATION

Several techniques and tools enable the emulation of CPU performance, giving the ability to test an application or an algorithm in an environment with a configurable and reproducible level of heterogeneity of processors. However, the emulation of multi-core (or multi-processor) systems has not been addressed until now. This work targets the emulation of several nodes within one physical node, each possibly with a given number of cores and emulated frequencies. The emulation should also be independent from the number of processes or threads composing the application.

A partial solution to this problem is provided by the Linux *cgroups* subsystem, which can be used to setup separate scheduling zones with a smaller number of cores. However, while this can be used to restrict a subtree of processes to a subset of CPU cores, the CPU frequency emulation must be carried out by different means.

In this section, three methods for multi-core CPU performance are presented. CPU-Hogs is a multi-core variant of the idea of using spinning CPU-intensive tasks to prevent other applications from using the CPU. Fracas leverages Linux scheduling to allocate the desired CPU share to applications, and CPU-Gov alternates between two hardware CPU frequencies to achieve the desired emulated frequency.

In addition to the ability to emulate CPU speed, these methods can perform it within different scheduling zones, or *virtual nodes*, with a different CPU speed and a subset of cores. Therefore, it is even possible to emulate a configuration with several *virtual nodes*, each of them having a different number of core and a different CPU speed (Figure 2).

A. CPU-Hogs

A basic method to degrade the perceived CPU performance is to create a spinning process that will use the CPU for the desired amount of time, before releasing it for the application. This was already implemented in Wrekavoc [5].

The CPU-Hogs method generalizes this idea to the multi-core case. One CPU burner thread per core is created, and assigned to a specific core using scheduler affinity. They are assigned the maximum realtime priority, so that they are always prioritized over other tasks by the kernel. The CPU burners then alternatively spin and sleep for configurable amounts of time, leaving space for the other applications during the requested time intervals.

However, creating one CPU burner per core is not enough in the multi-core case. If the spinning and sleeping periods are not synchronized between all cores, the user processes will migrate between cores and benefit from more CPU time than expected (Figure 3). This happens in practice due to interrupts or system calls processing that will desynchronize the threads. In CPU-Hogs, the spinning threads are therefore synchronized using a *POSIX thread barrier* placed at the beginning of each sleeping period.

This method is easily portable to other operating systems (and should be portable without any code change to other POSIX systems). It may have problems scaling to a large number of cores due to the need for frequent synchronization between cores, even though in practice we did not encounter any observable overhead with 8-core systems.

B. Fracas

Whereas CPU-Hogs is responsible for deciding when CPUs will be available for user processes, another solution is to leave that decision to the system scheduler (known as Completely Fair Scheduler since Linux 2.6.23), which is already in charge of scheduling all the applications on and off the CPUs. This is the idea behind Fracas, our scheduler-assisted method for CPU performance emulation
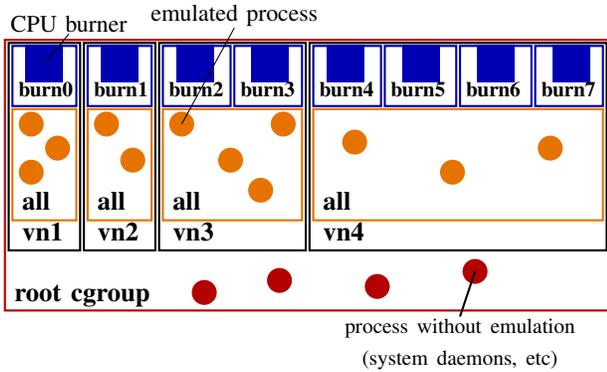
Figure 4: Structure of *cgroups* in Fracas for the example from Figure 2.

which shares many ideas with Krash [7]. Fracas was already presented in [8], but gained support for emulating several virtual nodes on a physical machine since then.

With Fracas, one CPU-intensive process is started on each core. Their scheduling priorities are then carefully defined so that they run for the desired fraction of time. This is implemented using the Linux *cgroups* subsystem, that provides mechanisms for aggregating or partitioning sets of processes or threads into hierarchical groups. As shown on Figure 4, one *cgroup* per virtual node is first created. Then, inside each of these *cgroups*, one *cgroup* named *all* is created to contain the emulated user processes for the given virtual node. Finally, *cgroups* are created around each of the CPU burner processes.

Additionally, within each virtual node priorities of *all* (pr_{all}) *cgroup* and every *burn* (pr_{burn}) *cgroup* must be properly adjusted. The CPU time is distributed proportionally to the priorities of *cgroups*, thus the values are set so that the following formula holds:

$$\frac{pr_{all}}{pr_{all} + pr_{burn}} = \mu$$

where μ is a ratio of the emulated CPU frequency to the maximum CPU frequency. In particular, when the virtual node emulates a CPU half as fast as the physical CPU ($\mu = 0.5$), then both the priorities will have the same value.

Since the system scheduler is responsible for switching between the CPU burner process and the user processes, the frequency of those context switches will influence the quality of the resulting emulation. Two scheduler parameters tunable in `/proc/sys/kernel/` have been experimentally verified to influence the scheduling granularity and the quality of emulation: `sched_latency_ns` and `sched_min_granularity_ns`. Their default values provide results of acceptable quality as shown in [8], nevertheless they are both configured to their lowest possible values (0.1 ms) for the experiments described in the following sections.

It is worth noting that the implementation of Fracas is strongly related to the Linux kernel's internals: as the scheduling is offloaded to the kernel's scheduler, subtle changes to the system scheduler can severely affect the correctness of Fracas. Results presented in this paper were obtained using Linux 2.6.33.2, but older kernel versions (for example, version 2.6.32.15) exhibited a very different behavior.

C. CPU-Gov

Contrary to the previous methods where the emulation is done purely in software, CPU-Gov is a hardware-assisted approach. It leverages the hardware frequency scaling to provide emulation by switching between the two frequencies that are directly lower (f_L) and higher (f_H) than the requested emulated frequency (f). The time spent at the lower frequency (t_L) and at the higher frequency (t_H) must satisfy the following formula:

$$f = \frac{f_L t_L + f_H t_H}{t_L + t_H}$$

For example, if the CPU provides the ability to run at 2.0 GHz and 2.4 GHz, and the desired emulated frequency is 2.1 GHz, CPU-Gov will cause the CPU to run 75% of the time at 2.0 Ghz, and 25% of the time at 2.4 Ghz. The frequency of switching is configurable, but was set to 100 ms during the experiments presented in the following sections.

As described, CPU-Gov can only emulate frequencies which are higher than the lowest provided by hardware: a different solution is required to emulate frequencies that are lower than the ones provided by *frequency scaling*. For those, a virtual *zero frequency* is created by stopping all the processes in the virtual node. For this, the Linux *cgroup freezer* is used, which has the advantage of stopping all tasks in the *cgroup* with a single operation.

This method has the advantage that, when the frequency is higher than the lowest frequency provided by hardware frequency scaling, the user application is constantly running on the processor. Hence, its CPU time will be correct, which is not the case for the other methods.

However, this method suffers from the limitation mentioned in Section II about frequency scaling: on some CPUs, it is not possible to change the frequency of each core independently: some cores might have to be switched together for the change to take effect, due to the sharing of caches, for example. This is taken into account when allocating virtual nodes on cores, but limits the possible configurations. For example, on 4-core CPUs, it might not be possible to create 4 virtual nodes with different emulated frequencies. To get information about the topology and relations between cores, `hwloc` library is used.

Another disadvantage is that this method relies on the frequencies advertised by the CPU. On some AMD CPUs, some advertised frequencies were experimentally determined

to be rounded values of the real frequency (the performance was not growing linearly with the frequency). It would be possible to work-around this issue by adding a calibration phase where the performance offered by each advertised frequency would be measured.

IV. EVALUATION WITH MICRO-BENCHMARKS

In this section, CPU-Hogs, Fracas and CPU-Gov are evaluated together with the CPU emulator provided in Wrekavoc [5], i.e., CPU-Lim, which was already described in Section II.

The original CPU-Lim from Wrekavoc was rewritten for those experiments, as it was not usable under its current form. First, support for controlling virtual nodes was added. Whereas the original CPU-Lim tracks processes of a specific user, the modified version controls processes inside a given *cgroup*. Moreover, the modified version uses more accurate interfaces to query process timers, improving the quality of emulation. Other modifications were also considered, like the use of the Linux *cgroup freezer* feature instead of signals to stop and resume tasks, and the use of better interfaces to retrieve the CPU time on a per-thread basis, but it was chosen to avoid diverging from the original design.

All these methods were tested with 6 different micro-benchmarks, each representing a different type of workload:

- CPU-intensive – a tight, CPU-intensive loop is performed for 3 seconds. The result is the computation rate.
- IO-intensive – UDP datagrams of size 1 KB are sent to a non-existing IP address for 1 second. The result is the communication rate (number of send loops per second).
- Computing and sleeping – a CPU-intensive loop is executed for 1 second, then the process sleeps for 1 second (as if it was waiting for IO operations to finish, or for synchronization with other processes), and finally the CPU-intensive loop is executed for 1 second again. The result is a number of loops performed during the test divided by the time needed to execute them (computation rate). Since the sleeping time is independent from the CPU performance, the result of this benchmark is expected to grow linearly with the CPU performance.
- Memory speed – the classic STREAM benchmark is used to measure (in MB/s) the sustainable memory speed.
- Multiprocessing – 5 processes are created and each of them runs an instance of CPU-intensive benchmark for 1 second. After that, they are all joined (using `waitpid` call) and the result is a number of loops performed by each of the tasks (which is the same for all of them) divided by the time between the processes' creation and the completion of the final joining. If all

the tasks can run simultaneously, the result should be the same as the result of CPU-intensive benchmark.
- Multithreading – a modification of the multiprocessing benchmark: instead of processes, POSIX threads are created and managed. All other details remain the same. In order to ease the reproduction of the experimental results, the source code and scripts used to performed experiments is made available[1].

A. Experimental Methodology

The tests presented in this section were run on a cluster of 25 identical Sun Fire X2270 machines, equipped with two Intel Xeon X5570 (Nehalem microarchitecture) and 24 GB of RAM each. The Intel Xeon X5570 provides frequency scaling with 11 different levels: 2.93, 2.80, 2.67, 2.53, 2.40, 2.27, 2.13, 2.00, 1.87, 1.73, and 1.60 GHz. Both *Intel Turbo Boost* and *Hyper-Threading* were disabled during the experiments.

An unmodified 2.6.33.2 Linux kernel was used on the nodes. To schedule the tests, a test framework written in Python was developed. Only one test was running on each node at a given time. Each individual test was reproduced 40 times and the values presented on the graphs are the average of all samples with the 95% confidence intervals (though most experiments produce very stable results, hence the confidence intervals might not be visible). The same tests were also run on a cluster equipped with AMD Opteron 252 CPUs, and no significant difference was found.

B. Benchmarks on One Core

This section describes the results obtained by the micro-benchmarks described in the previous section on a virtual node containing only one core. CPU-Lim, CPU-Hogs, Fracas and CPU-Gov are evaluated, and results obtained using only hardware frequency scaling (later described as *CPU-Freq*) are also included for comparison.

1) CPU-intensive workload: As can be seen in Figure 5a, all methods perform well when a CPU-intensive application runs inside the emulated environment, i.e., they all scale the speed of the application proportionally to the value of emulated frequency. However, though it cannot be seen on the graphs, the most stable results are produced by Fracas method, and the results with the highest variance are produced by CPU-Lim method.

2) IO-intensive workload: How the emulation of CPU frequency should influence the performance of the network, or of any other IO operation, is unclear. One could assume that their respective performance should be completely independent. However, IO operations require CPU time to prepare packets, compute checksums, etc. The methods exhibit very different behaviors, as shown in Figure 5b, though which one should be considered the best is not clear.

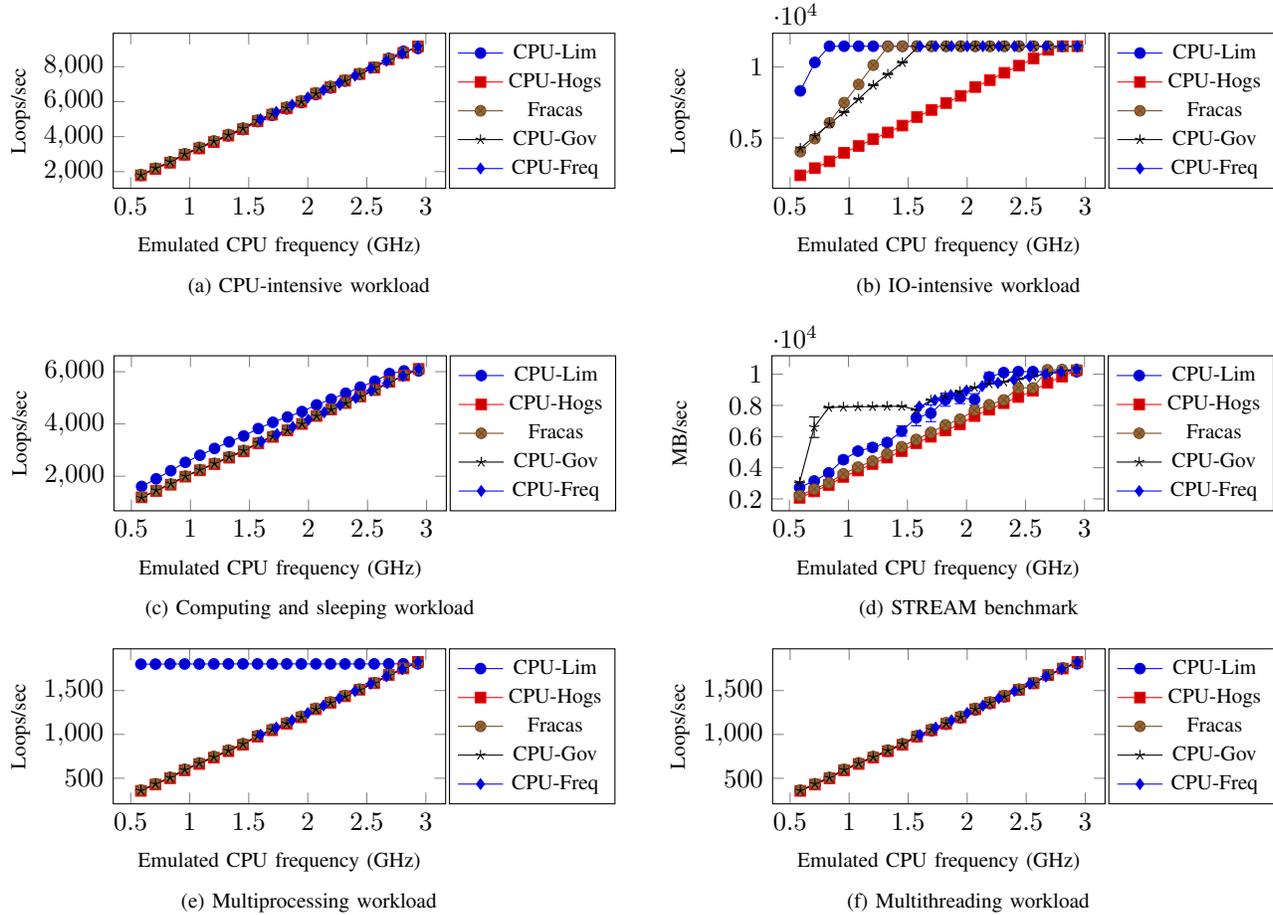[1]http://www.loria.fr/~lnussbau/files/cpuemul-exp.tgz

Figure 5: Results for micro-benchmarks running on one core (Section IV-B)

CPU-Lim, Fracas and CPU-Gov only scale IO performance up to a certain point, which could be consistent with the fact that IO operations require a certain amount of CPU performance to perform normally, but that adding more CPU performance would not improve the situation further. On the other hand, CPU-Hogs scales IO performance linearly with the emulated frequency.

3) CPU and sleeping workload: All the methods, with a sole exception of CPU-Lim, perform very well in that benchmark, as presented in Figure 5c. This is expected, because CPU-Lim does not take the sleeping time of a process into the account, and wrongly gains an advantage after a period of sleep.

Amongst the well-behaving methods, the most stable results are produced by CPU-Gov.

4) Memory speed: How the memory speed should be affected by the CPU speed emulation is not clear, since both parameters are closely related. The speed of the memory directly controls the speed of instruction fetching and of memory accesses, and consequently the speed of execution. On the other hand, a slower CPU will execute instructions

accessing the memory slower, so will indirectly degrade the memory speed, at least from the user perspective.

Therefore, which method provides the *best* results is not obvious in Figure 5d. It may be only noted that the most predictable and easy to understand behavior is that of CPU-Hogs and Fracas, since they are stable and almost linear with the respect to the emulated frequency. This can neither be said about CPU-Lim method, whose results fluctuate greatly, nor about CPU-Gov method which gives predictable results, but without any obvious relation to the value of emulated frequency.

5) Multiprocessing: With multiple tasks, either processes or tasks, it is expected to linearly and independently degrade the speed of each CPU-intensive task. Most methods provide good results as seen in Figure 5e, with the exception of CPU-Lim. As CPU-Lim computes the CPU usage independently for each process, but does not sum it to compute the *virtual node*'s CPU usage, it appears that the CPU-Lim method does not emulate anything, as the CPU usage of each independent process stays under the limit.

6) Multithreading: The expected behavior in the benchmark presented in Figure 5f is exactly the same, as in the previous benchmark. This time even the CPU-Lim method performs very well, because the CPU time of the emulated threads is accumulated for the whole process. Again, there is no clear winner in terms of the stability of the results, i.e., all methods give satisfactory results in that sense.

C. Benchmarks on 2, 4 and 8 Cores

Contrary to the previous set of tests, the micro-benchmarks were run in an environment emulating more than one core. In each case, five user processes or threads are executed. All single-task benchmarks gave the same results as before, so they are not included.

1) CPU-Hogs and CPU-Gov: The results (Figure 6) clearly show the superiority of CPU-Hogs and CPU-Gov methods, as the result of the benchmarks is proportional to the emulated frequency only in their case (and for the CPU-Freq method, but it's not able to emulate continuous range of frequencies, and therefore is not considered as a fully functional method). Additionally, CPU-Hogs is superior to CPU-Gov in terms of stability of results, providing results with a slightly smaller variation.

2) CPU-Lim: The CPU-Lim method is able to properly emulate multiprocessing type of work, however only when each process can run on an independent core. This can be seen in Figure 6e – all processes cannot saturate available cores and CPU-Lim works as required. Nevertheless, we can see that the result is too high most of the time for benchmarks with a lower number of tasks, due to CPU-Lim's problem with computing CPU time (Section II). As the benchmark consists of 5 processes, each of them will get approximately $\frac{2 \cdot 100\%}{5} = 40\%$ and $\frac{4 \cdot 100\%}{5} = 80\%$ of the CPU time, for cases in Figure 6a and Figure 6c, respectively. As can be seen, this is precisely a fraction of maximum CPU frequency where the graph suddenly drops. Therefore, in general, CPU-Lim will not properly emulate a group of processes in multi-core configuration.

A different problem can be seen in the case of the multithreading benchmark. Now, the CPU-Lim method gives values lower than the expected ones. This is because it controls processes (or groups of threads), not threads. The CPU time of a process is a sum of CPU-times of all its threads, and as such, it may go up faster than the realtime clock. Moreover, when CPU-Lim sends a signal to stop the process, all its threads will be stopped. Put together, this explains why the results in Figure 6b, Figure 6d, and Figure 6f are precisely 2, 4, and 8 times lower than those for CPU-Hogs or CPU-Gov.

A very strange phenomenon can be observed in Figure 6c – the benchmark gives higher results in the environment emulated with CPU-Lim than in the unmodified one. This counterintuitive behavior is due to the kernel which, when the processes are run normally, will put every process on one of 4 cores and, as there are 5 processes in total, one core will execute two processes simultaneously. They will run twice as slow as the remaining ones and, consequently, will degrade the overall result of the benchmark (it is possible to mitigate the problem by extending the time of the benchmark). With CPU-Lim method, the processes are stopped periodically, forcing the scheduler to migrate them between unused cores and giving them fairer amount of CPU time. It seems that the Linux scheduler, as much as advanced it is, is by no means perfect. But even knowing that, the conclusion must be drawn that CPU-Lim behaves improperly, as we aim to emulate the exact behavior of the unmodified kernel.

3) Fracas: The results for Fracas method show, as was already observed in [8], that the method does not work well for multitasking type of work. The results of the benchmarks are much lower than expected. For example, it can be seen in Figure 6b, Figure 6d and Figure 6f that the results are 2, 4, and 8 times lower, respectively. This is because the priority of *cgroup* consisting of the emulated tasks is constant (as defined by a formula in Section III-B). Even if the emulated tasks are running on different cores, the total allowed CPU time of them will be bounded by this priority. The priority of the *cgroup* can be adjusted so that it will work for a particular number of processes inside the emulated environment, but, unfortunately, there is no a *generic* value that will work for every possible number of tasks.

Also, one can see a significant discrepancy between pairs of figures: Figure 6a and Figure 6b, Figure 6c and Figure 6d. This does not happen in the last pair: Figure 6e and Figure 6f. Again the reason is the scheduler and was observed in CPU-Lim case before - when there are more tasks than cores in the system, some arbitrary decisions made by the system make the parallel execution suboptimal. Evidently, this is much more expressed in the case of multiple number of threads, not processes, but was also manually triggered in the latter case. This could have been expected, but the difference in the execution time is startling. More confusingly, the behavior of the scheduler can change quite dramatically with every version of the Linux kernel. That was in fact so, and other anomalies were observed with different releases of Linux kernel.

The results clearly show that reliance on the system scheduler may be deceiving and, as a result, the Fracas method should not be used to emulate an environment with multiple tasks (unless the number of cores is greater than their number).

V. CONCLUSIONS

Emulation of CPU performance is an important asset in the context of the evaluation of applications targeted at heterogeneous systems. In this work we propose three new methods for the emulation of CPU performance: Fracas (based on a preexisting idea from the Krash workload

(a) Multiprocessing on 2 cores

(b) Multithreading on 2 cores

(c) Multiprocessing on 4 cores

(d) Multithreading on 4 cores

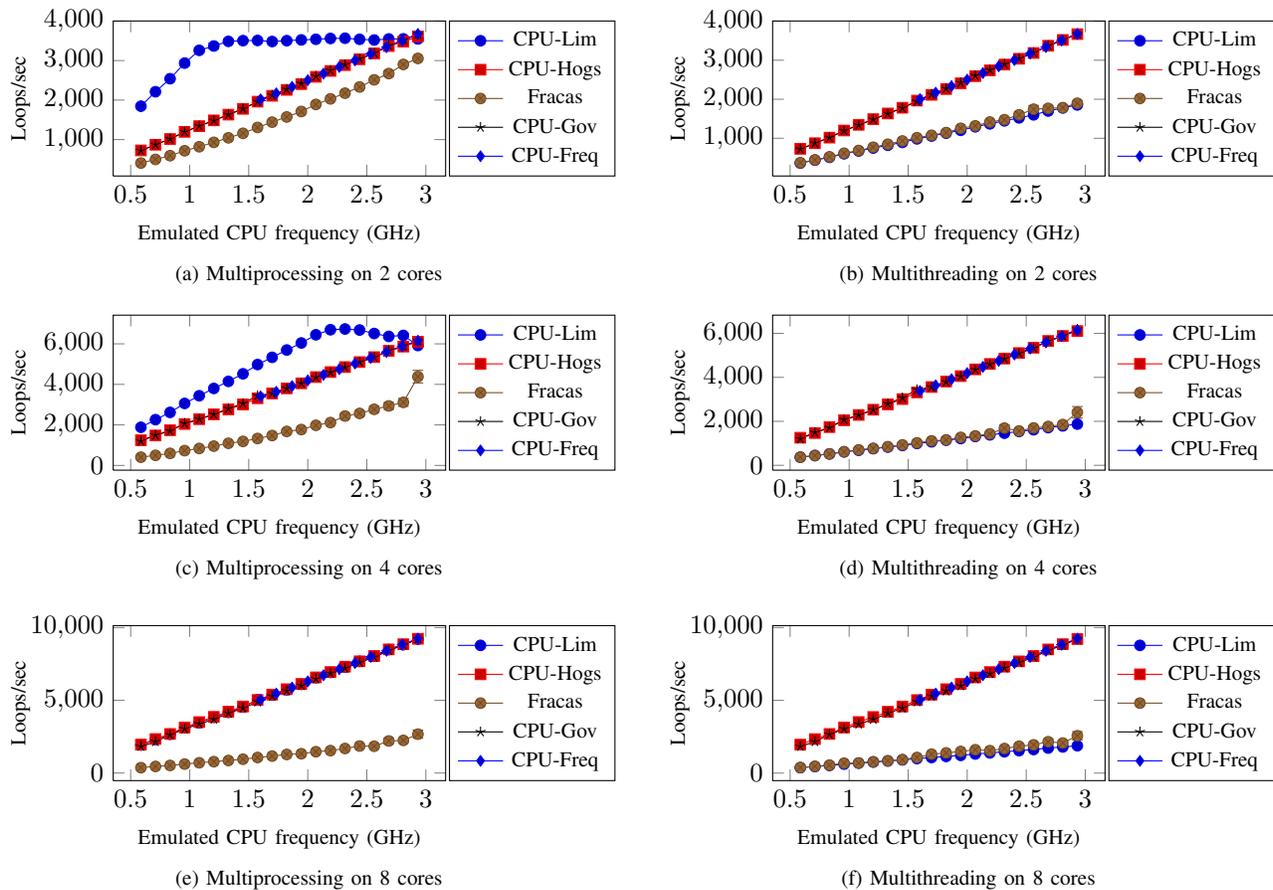(e) Multiprocessing on 8 cores

(f) Multithreading on 8 cores

Figure 6: Results for micro-benchmarks running on 2, 4 and 8 cores (Section IV-C)

injection tool), CPU-Gov (based on the hardware CPU frequency emulator), and CPU-Hogs (a multi-core CPU burn implementation).

After a detailed description of those three methods, we compare them with the current CPU emulator in Wrekavoc, i.e. CPU-Lim, by running a carefully designed set of micro-benchmarks. Results show that both CPU-Gov and CPU-Hogs generate the desired experimental conditions, with some variations in some benchmarks where the expected behavior is not completely clear. In contrast, CPU-Lim and Fracas exhibit severe limitations, especially when used to execute applications consisting of several processes or threads in a multi-core environment.

In the future, we will continue to evaluate those CPU emulators by running experiments with real applications. We also plan to experiment with memory performance emulation, as memory speed has become a very important parameter for understanding the performance of applications.

REFERENCES

[1] J. Gustedt, E. Jeannot, and M. Quinson, "Experimental validation in large-scale systems: a survey of methodologies,"
Parallel Processing Letters, vol. 19, pp. 399–418, 2009.

[2] H. J. Song *et al.*, "The microgrid: a scientific tool for modeling computational grids," in *SuperComputing*, 2000.

[3] A. Vahdat *et al.*, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 271–284, 2002.

[4] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," in *OSDI'02*, Boston, MA, 2002.

[5] L.-C. Canon, O. Dubuisson, J. Gustedt, and E. Jeannot, "Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool," *Journal of Systems and Software*, vol. 83, pp. 786–802, 2010.

[6] F. Broquedis *et al.*, "hwloc: a generic framework for managing hardware affinities in HPC applications," in *PDP*, 2010.

[7] S. Perarnau and G. Huard, "Krash: reproducible CPU load generation on many cores machines," in *IPDPS '10*, 2010.

[8] T. Buchert, L. Nussbaum, and J. Gustedt, "Accurate emulation of cpu performance," in *HeteroPar*, 2010.

[9] D. Gupta, K. V. Vishwanath, and A. Vahdat, "Diecast: testing distributed systems with an accurate scale model," in *NSDI*, 2008.