KRASH: Reproducible CPU Load Generation on Many-Core Machines

Swann Perarnau, Guillaume Huard
INRIA Moais research team, CNRS LIG lab.
Grenoble University, France
Email: {perarnau,huard}@imag.fr

***Abstract*—In this article we present KRASH, a tool for reproducible generation of system-level CPU load. This tool is intended for use in shared memory machines equipped with multiple CPU cores which are usually exploited concurrently by several users. The objective of KRASH is to enable parallel application developers to validate their resources use strategies on a partially loaded machine by *replaying* an observed load in concurrence with their application. To reach this objective, we present a method for CPU load generation which behaves as realistically as possible: the resulting load is similar to the load that would be produced by concurrent processes run by other users. Nevertheless, contrary to a simple run of a CPU-intensive application, KRASH is not sensitive to system scheduling decisions. The main benefit brought by KRASH is this reproducibility: no matter how many processes are present in the system the load generated by our tool strictly respects a given load profile. This last characteristic proves to be hard to achieve using simple methods because the system scheduler is supposed to share the resources fairly among running processes.**

Our first contribution is a method that cooperates with the system scheduler to produce a CPU load that conforms to a desired load profile. We argue that this cooperation with the system scheduler is mandatory in the generator to reach a good reproducibility, a high precision and a low intrusiveness. Taking advantage of Linux kernel capabilities, we implemented this method in KRASH (Kernel for Reproduction and Analysis of System Heterogeneity). We have run experiments that show that KRASH provides a precise reproduction of the desired load and that it induces a very low overhead on the system. Our second contribution is a qualitative and quantitative study that compares KRASH to other tools dealing with system-level CPU load generation. To our knowledge, KRASH is the only tool that implements the generation of a dynamic load profile (a load varying with time). When used to generate a constant load, KRASH result is among the most realistic ones. Furthermore, KRASH provides more flexibility than other tools.

I. Introduction

New hardware architectures, composed of multiple cores and a shared memory, spread in the domain of High Performance Computing. These many-core machines are usually used as computing servers and shared by several users. In this context, available resources to a given user are only a part of the shared machine: processes from other users constantly start or exit, acquiring or releasing resources. As the operating system tries to dispatch fairly those resources among running processes, programs are faced with heterogeneous resource availability. This heterogeneity presents two dimensions: different CPU cores do not share the same load and these loads vary with time.

To cope with this dynamic heterogeneity, new parallel programming paradigms have been put forward: fine grain tasks parallelization possibly combined with work stealing [1] or adaptive algorithms [17]. When comparing these different approaches, one might consider various evaluation criteria such as how they compete in the system to get resources control or how efficiently they use the given resources they are given. In this article we do not address the evaluation of a parallel application capabilities to compete for resources access. The work we present is an experimental testbed able to control resources unavailability (what we name load) according to a determined pattern. Consequently, this work makes possible to compare the way distinct parallel programming paradigms use a fixed set of heterogeneous resources (in other words the same experimental conditions). This efficiency comparison is usually difficult as most system schedulers try to fairly balance resources access among running processes. In this context, a fine grained parallel application that spawns more processes than actual computing resources will get more access opportunities than other parallelization schemes, making the efficiency comparison impossible.

A real production machine cannot be used for such a comparison: its load constantly varies due to other users, resulting in an environment not consistent across experiments. One might think about doing the comparison using simulation tools such as SimGrid [6]. But, as pointed out by Franck Cappello et al. [2], the simulation of a complex system usually fails to reproduce its entire behavior. Therefore, a detailed and complete efficiency comparison of those new algorithms will eventually require their execution on a real machine with a controlled dynamic environment. To precisely produce and reproduce this controlled dynamic environment, a specific tool is needed: a load generator. This tool must generate the same desired concurrent load, whatever the parallel program under evaluation, no matter how many processes it creates or the system resources it requires. In this article we focus only on CPU load, which is usually the most important element in high performance computing. Of course, other system resources might have a significant effect on applications performance but their importance in a complete load generation would be biased if the CPU load generation is not sufficiently precise. Besides, we plan to study the generation of loads on other system resources such as cache and memory in our future works.

Other methods to generate CPU load for parallel algorithms validation purposes have been discussed in the literature [15], [5]. We will show later that these solutions, as well as less elaborate ones, do not satisfy all our objectives.

Indeed generating a reproducible CPU load is not as simple as running some CPU intensive application in a concurrent process. Because of the scheduler efforts to balance resources usage between running processes, the resulting load will depend on the overall state of the machine. In this paper we present a new method to generate a CPU load that matches a given load profile whatever the target machine state (number of running processes or already allocated resources). Our method cooperates with the system scheduler to produce the desired load without changing its usual policy. As we will see in the experiments, maintaining the same scheduling policy for remaining available resources given to running applications is the only way to avoid obtrusiveness caused by the generator. We implemented this method in a tool named KRASH (Kernel for Reproduction and Analysis of System Heterogeneity). This implementation is based on recent Linux kernels capabilities that enable our load injector to behave as determined by our method and to maintain a very low intrusivity.

In the next section, we present basics of system schedulers behavior and a CPU load model. We then present, in Section III, our method for reproducible CPU load generation and its implementation in KRASH. In Section IV, we validate our implementation regarding realism and reproducibility and we compare it to existing tools that serve the same purpose. Finally, before concluding in Section VI, we discuss related works that are not directly applicable to our problem.

II. Basics of System Scheduling and CPU Load

In this section, we remind basic definitions of scheduling. We also provide a clear definition of a CPU load and the resolution of CPU load generators.

To enable the concurrent execution of several programs the classical approach is time-slicing: time intervals of exclusive execution on the CPU are assigned to the various applications running in the system. On many-core systems this approach is simply extended by running a distinct scheduler on each CPU core and by sharing processes among schedulers. All modern operating systems provide this timeslicing. For instance, the Linux kernel calls these timeslices *jiffies*. Depending on the configuration, between one hundred and a thousand jiffies occur each second in the system.

Definition 1. *The load of a CPU core during a time interval is the proportion of this time interval during which the CPU core has been assigned to some application or used by the system (time of unavailability). We call this definition the absolute load of the CPU core. We also define the load for an application as the proportion of the time interval during which the CPU core has been assigned to this application. Absolute load is computed by the ratio:*

$$absolute\ load = \frac{timeslices\ of\ unavailability}{total\ timeslices\ of\ the\ time\ interval}$$

The assignment of CPU cores to applications on a timeslice basis and the definition of load *during* a time interval induce several issues about load measurement. First, the measurement of a CPU core load is usually done by a monitoring program that competes for execution on the same CPU core. Thus, measurements of the CPU core load at a frequency which is higher than the number of timeslices per second is not possible. In other words, the timeslicing mechanism enforces an absolute limit to the load monitoring precision. Second, classical schedulers existing on modern systems dispatch the timeslices among applications following a fair share principle: each application should obtain a computing time depending on its priority. Because access to a resource is given as whole timeslices, this policy can only be enforced asymptotically. Thus, a relatively large period of time is required to achieve fair sharing (all running processes have been allocated enough execution time on the CPU). Therefore, load measurements made with a periodicity that is lower than this large period of time are likely to lead to false results (not matching the actual distribution of the processor between applications performed by the scheduler).

These considerations lead us to the definition of scheduler resolution. The scheduler resolution of a system is the minimum period of time required by the scheduler in activity to fairly share the CPU core among running applications. As scheduling can only asymptotically converge to a perfect balance in resources assignment to applications, the resolution is generally chosen by convention. This is the time interval necessary to usually reach a sufficiently good share of the resources. For instance, the resolution which is commonly used by system measurement tools (`top`, `htop`) for the Linux kernel is one second. As a side effect, measurement of the load on a CPU core cannot be performed more frequently than the scheduler resolution without leading to biased results. Similarly, to be meaningful, the generation of a given load on a CPU core will only be performed at a resolution superior or equal to the system scheduler resolution. Load profiles will also be accordingly defined:

Definition 2. *We define a CPU load profile of a CPU core during a given time interval as a piecewise constant function that represents the temporal evolution of the load on this CPU core. In this function, each piece is at least the same size as the system scheduler resolution. By extension, a CPU load profile during a given time interval is the set of its profiles (one for each core).*

In this article, we will not discuss methods to obtain such a load profile. Several solutions exist, namely measurement using standard system tools, synthesis with statistical models [13] or simulation traces.

III. Reproducible Generation of CPU load

In this section, we present the first contribution of this article, a method for generating a reproducible CPU load along with its implementation for recent Linux kernels (2.6

series). Beside reproducibility, we aim at designing a precise and unobtrusive generator.

A. General Methodology

Ensuring the reproducibility of a load generation is not an obvious task. It requires the constant monitoring of the CPU assignment decided by the scheduler to adjust the generated load. This monitoring can be performed in several ways.

The first idea which comes to mind is to create a CPU intensive process that adjusts its own priority to reach the expected load. A CPU benchmark such as [7], [10] could be chosen and instrumented for this task. Unfortunately, this approach lacks precision: the resulting load both depends on the priority of the CPU intensive process itself and on the other processes present in the system. Nothing guarantees that the CPU intensive process will be executed at the scheduler resolution to adjust its own priority. This is a major problem when generating low loads: the load process having a low priority, it is not awaken often enough. Moreover, this poor precision will also result in poor reproducibility. We should also mention that very high loads are not always achievable using this method. When many other processes are present in the system, even if the loading process priority alters the share balance, the scheduler still tries to periodically assign the CPU to all processes (at least for one timeslice). This results in a load that cannot get as close as desired to 100%.

A more elaborate scheme for generating load is to create a special distinct process, the supervisor, which will always be executed in priority on a regular basis. The duty of the supervisor is to adjust the priority of the CPU intensive loading process. If the supervisor is executed at a frequency that matches the scheduler resolution, it will be able to adjust the injected load at a rate that ensures a good precision (as good as possible if we take into account the arguments presented in Section II). The main drawback of this method is its potential intrusiveness: the supervisor itself, because it is executed very frequently, will induce a load in the system. To ensure precision, this load has to be minimized or taken into account as a part of the load profile that should be reproduced. Nonetheless, it will always prevent the generator to generate extremely low load levels.

A variant of this supervisor-based load injection, used in the current version of Wrekavoc [15], monitors the system to stop and restart application processes. That way, the CPU is not directly loaded by the generator, but the application perceives a CPU unavailability during the time it is stopped. The drawback of this method, as we will see in the evaluation section, is that it produces undesirable side effects regarding the scheduler and its policies on I/O events. Furthermore, this approach does not solve the intrusiveness issue related to the supervisor (which is still required).

Our solution is to directly cooperate with the scheduler to avoid previous issues. The principle is quite simple: place on each CPU core a CPU intensive process that will act as a load generator. Then, act directly within the scheduler and at scheduler resolution to assign to our load generator the desired proportion of available timeslices. With this method, the generation is precise, because it is performed at scheduler resolution. Moreover, it is unlikely to be intrusive, because the generator is considered by the scheduler, at the same time as other processes, when ensuring fair resources share. Alas, this method requires a direct interaction with the scheduler. Fortunately, as we will see in the next section, some schedulers provide facilities to implement our method.

B. Generator Implementation for Linux

We implemented our load generation method for Linux by taking advantage of capabilities added in its recent kernel releases ($\geq$ 2.6). The two features which our generator depends on are the `cpuset` and `group scheduling` mechanisms. Before we present our solution, a short description of these two features seems to be required.

The `cpuset` feature enables programmers to attach to processes a set of CPU cores on which they can be executed. This restriction to a set of resources is enforced by the kernel itself, which places processes into the internal list of a matching scheduler and migrates them when necessary.

The `group scheduling` feature enables programmers to control resources sharing performed by the scheduler. When this feature is in effect, all processes necessarily belong to one control group, possibly the default group if no other group has been specified, and all groups have a priority. The scheduler distributes timeslices between groups in proportion to their priority. Within each group, classical priority scheduling is applied between processes to share assigned timeslices. Among interesting characteristics, we should mention that groups priority can be changed dynamically using a virtual filesystem, that `cpuset` restrictions can be applied to a whole group, and that processes inherit group membership. We should also mention that timeslices distribution among groups is performed by the scheduler according to the groups hierarchy defined by the virtual filesystem structure. Nevertheless we will not use this last characteristic as all our groups will be created at the same hierarchy level (which is just under the *root* group).

We have used these features to implement our method for load generation. Our generator is set in place by performing the following steps:

- create a new control group that we name the *base* group and move all processes into this group.
- for each CPU core, create a load generation process (we name it the *boulder*), a new control group, attach the load generation process to this control group and restrict the control group to the concerned CPU core.
- choose a CPU core to run a supervisor process. This supervisor is given an input load profile and monitor load on all the load generation processes. Whenever required, it adjusts the priority of the groups that contain load generation processes.

Figure 1 shows a simple diagram of this method. The first step of our implementation is just meant to confine all existing processes (and their possible future children) into a single

group at the same hierarchy level as the other groups we create. The *boulder* processes are simple infinite loops that render their CPU core unavailable during their timeslices. This simplicity ensure minimal cache and memory footprints. Thus, the *boulders* do not interfere much with the rest of the system.

The supervisor process is more elaborate. It relies on the specification of groups scheduling in Linux: timeslices are divided between groups in proportion to their priority. This is slightly different than priority scheduling which is implemented by giving more or less timeslices to processes depending on both their priority and their resources usage. In this last case, all processes will be given some timeslices and the repartition is not necessarily in proportion to processes priority. In fact, the proportion of timeslices given to a process will also depend on the number of other processes in the system. This issue does not exist with group scheduling if the number of groups in the system is stable. This enables us to precisely control timeslices repartition.

We should also mention some implementation tricks related to the supervisor. As this is a monitoring process, it should periodically determine if the current timeslices repartition is in conformity with the desired load. If this is not the case, it adjusts the priority of the groups containing the boulders as follows:

$$\text{new priority} = \frac{\text{base group priority}}{1 - \text{desired load}} \times \text{desired load}$$

This is a simple proportionality calculus which ensures that priorities proportions match the desired load. In principle, this adjustment should be performed on a regular basis matching the scheduler resolution. A naive implementation thus results in some intrusiveness due to the supervisor execution. Nevertheless, the adjustment is only required when the load generated by some boulder has to be changed. In other cases, if the groups organization and priorities are not modified, the scheduler will keep on dividing timeslices accordingly which will result in a constant load generation. With this idea in mind, we implemented the supervisor as a process that adjusts priorities and falls asleep immediately after for an unspecified time. Then, taking advantage of notifications within the Linux kernel we wake the supervisor process up only when a change is required by the load profile. This results in an extremely low intrusiveness which even enables the supervisor to run on one of the loaded cores without much harm.

IV. Evaluation

In this section, we validate our generator implementation. First, we present experiments that confirm that this implementation fulfills our requirements. Then we compare it to existing tools capable of some kind of CPU load generation. This comparison is both qualitative (offered features, flexibility of the generation) and quantitative (real world performance).

During all these experiments, we aim at evaluating several criteria that we consider of utter importance for any CPU load generator:

- reproducibility: the generator should be able to reproduce the desired load whatever the environmental conditions (others processes in the system, available memory, devices state).
- unobtrusiveness: the generator should not alter the qualitative behavior of the system. In particular, it should neither change the scheduler policy, nor produce additional performances degradation unrelated to CPU performance (I/Os degradation, for instance).
- precision and reactivity: we need a generator able to reproduce realistic load profiles in a very dynamic environment. In other words, the generator should be able to precisely produce the desired load with good reactivity.
- unintrusiveness: the additional load induced by the generator process should be very moderate. This characteristic is usually related to unobtrusiveness (an intrusive generator becomes often obtrusive) and versatility (a lightweight generator does not require extra resources for the supervisor process and can generate a wider range of different loads).

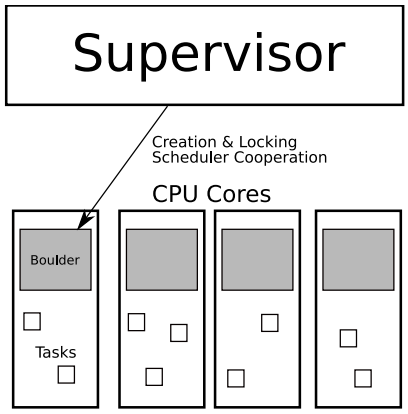


Figure 1: Diagram of our solution, based on boulder and supervisor processes

In this whole section, the machine used for experiments is a SMP system made of 8 Dual Core AMD Opteron 875 (2.2 GHz), 32 GB of RAM and a RAID 1 250 Go storage subsystem. This machine runs a Linux Debian Sid (unstable) Operating System with a recent kernel (version 2.6.30). This Linux kernel is configured with control groups and group scheduling activated. In some of our experiments we only used a subset of the available CPU cores. Unless specified differently, those cores are chosen in the logical order given by the kernel: if the experiment runs on 8 cores then the first 8 are used (only 4 CPUs are working). In our system, only the first two CPUs are dedicated to I/O. In all the experiments presented in this section, average values are calculated from the results of 30 consecutive runs.

Our load measurements are made using Sysstat [12]. This tool has a resolution of one second, it is comparable to ProcPS [4] which uses the same technique to gather system informations. This resolution is standard among Linux measurement tools. It is enough for the evaluation of CPU intensive applications and prevents the measurement tool for becoming too intrusive. Using such a tool, we gather a load

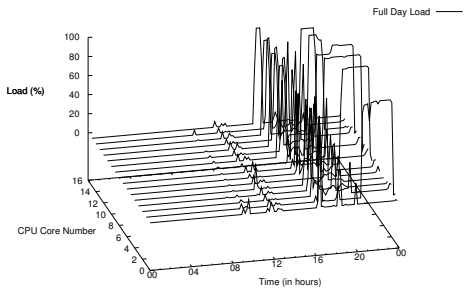


Figure 2: Example of a daily CPU load on our SMP System (16 cores)

profile such as the one presented in Figure 2 which is a load observed on our experimentation machine during a day. Besides, we also generated random load profiles using the modeling tools [13]. Using this kind of profiles has led us to the same conclusions. For the sake of clarity, we only present experiments made on 'real world' load profiles in the remaining of this section.

A. Generator Validation

With our first experiments, we start by validating KRASH precision, reactivity, unintrusiveness and reproducibility. We will address unobtrusiveness concerns in subsequent sections.

For this validation, we begin by creating a realistic load profile that we will later reproduce with KRASH. This load profile is created by running a reference application (an instance of Linpack [9]) and launching other processes during its execution to mimic a dynamic and shared environment.

The exact scenario is the following: execute our reference application on 8 cores and with default priority, launch a realtime priority task (infinite loop) during 15 seconds then kill it, do nothing during 10 seconds, run an instance of the NAS EP [7] benchmark on 8 CPU cores during 20 seconds (then kill it); do nothing during 16 seconds; start a new instance of the NAS benchmark on 8 CPU cores with a lower priority[1]. Once the last process finishes (30seconds later), start another instance of NAS on 8 CPU cores and, finally, stop it 20 seconds later. At this point Linpack is running alone on the system.

By monitoring the CPU attribution of Linpack, we obtained the opposing load profile in Figure 3(a) (that we name the reference profile). Notice that every application used to mimic a dynamic environment has a very low cache and memory footprints and only their CPU occupation hinders the Linpack computation.

The first validation experiment consists in replacing the previous realistic dynamic environment by KRASH (giving it the reference profile as input) and executing Linpack again. If the load is correctly recreated by KRASH and the Linpack application is disturbed in the same manner, we should obtain the same opposing CPU load as our reference profile. Figure 3(b) reports the load we measured during this experiment (CPU load generated by KRASH is in plain lines) along with the reference profile (dashed lines). We can clearly see that these two profiles are very similar: the average error between the generation by KRASH and the reference profile is about 2% with a standard deviation of 1%. This confirms the precision and the reactivity of KRASH: the generated load is roughly the same as the one resulting from a realistic dynamic environment.

The second validation experiment aims at validating load reproducibility in KRASH: we want to check that KRASH generates the required load no matter how many applications we run concurrently. Thus, to simulate a different application (different CPU use and different parallel execution scheme), we replace the Linpack control sample with 10 concurrent NAS instances on 8 CPU cores. The load that KRASH produces in this second validation experiment appears in Figure 3(c). Once again, the expected reproduction by KRASH of our reference profile is of high quality: the average error between the generation and the reference profile is about 2% with a standard deviation of 1%. Consequently, this experiment shows that even when changing drastically processes (number, nature) that run in concurrence with KRASH, our tool is able to reproduce the desired load with the same precision.

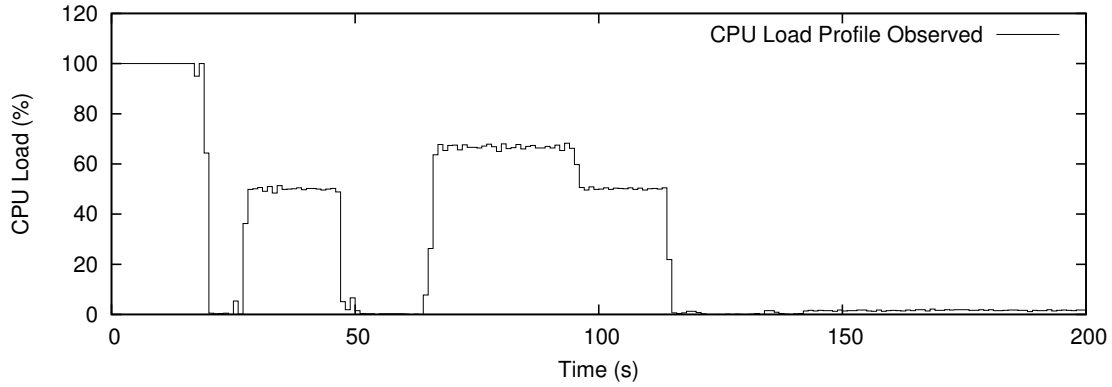
B. Qualitative Comparison with Other Solutions

In this section, we present existing solutions for CPU load generation and their differences with KRASH. Some of these solutions are intended for a more general use and include CPU load generation as part of their capabilities. For each one of these existing solutions, we discuss their characteristics, advantages and drawbacks in comparison to KRASH. A quantitative performance comparison presented in the next section completes this analysis.

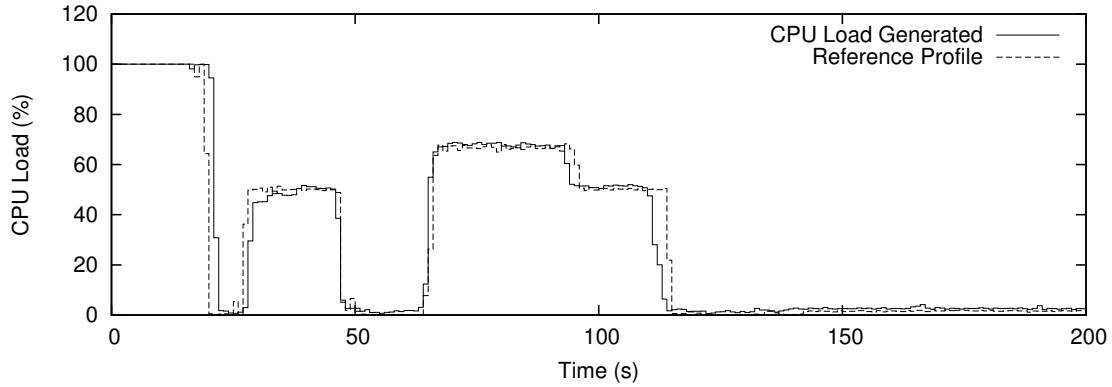
1) Wrekavoc: Wrekavoc [15] is a tool for heterogeneity simulation. Its objective is to enable users to limit the resources available to their application. The expected effect is to mimic an execution of the application on an heterogeneous platform. This tool can be used to limit CPU, memory and network use. Regarding only CPU, Wrekavoc is given a username and a desired CPU frequency as configuration. Then, for each process created by this user, it creates a supervisor process which inspects the `/proc` virtual filesystem on a regular basis to monitor the behavior of its associated user process. This supervisor stops or restarts its target process (using POSIX signals) to ensure that its CPU use is roughly equal to the ratio of desired CPU frequency to actual CPU frequency. To guarantee the periodic execution of the supervisor, it is run using real time priority scheduling.

Obviously, this method cannot be applied to applications that create more processes than CPU cores in the machine. Otherwise, the overall CPU use allowed to two processes exe-

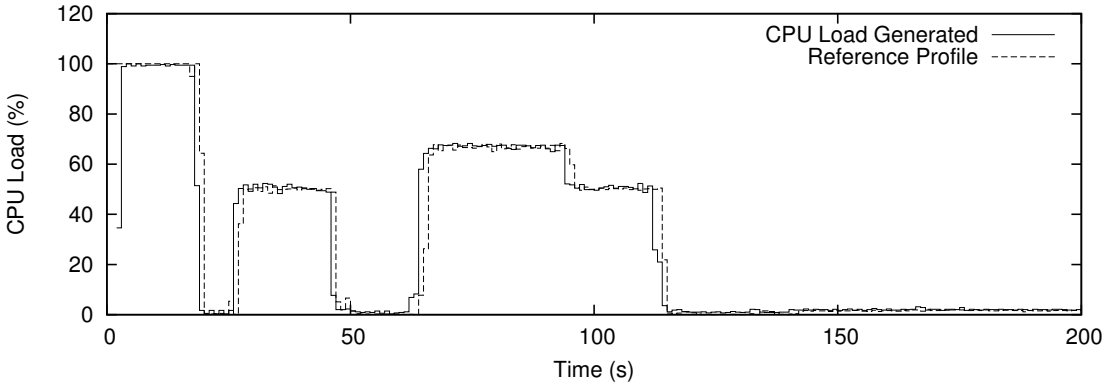
[1]On POSIX systems, a lower priority ensure more CPU attribution time to a process.



(a) Reference profile (realistic load) in concurrence with Linpack



(b) Load generated by KRASH in concurrence with Linpack



(c) Load generated by KRASH in concurrence with 10 NAS instances

cuting on the same core will not match the desired frequency: each one is independently limited, only based on its own resource consumption. This is not always an issue as most parallel applications do not create more processes than CPU cores. Nevertheless, this might happen when folding a parallel application designed for more resources than available. This makes Wrekavoc less versatile than KRASH. Furthermore, this method does not fit many-core machines: because of migration, processes might be assigned to distinct cores during their execution. Thus, Wrekavoc does not simulate platform heterogeneity but rather application heterogeneity. This issue is not addressed in their paper [15].

Regarding load profiles, Wrekavoc is only able to simulate constant loads. The desired simulated CPU frequency is given upon invocation of the tool and cannot be changed during its execution. Thus, Wrekavoc does not fulfill our objective of dynamic load profile generation on many-core machines.

2) Solutions based on real time priority scheduling: A direct way to generate a given load is to create a loading process scheduled in priority using a real time scheduling policy. This loading process just has to monitor its own

load and to adapt its activity to the desired load by running into some loop that simulates activity. This solution was implemented in a former release of Wrekavoc presented in their first paper [5]. This method acts completely from outside the scheduler and produces the desired load as a long CPU unavailability period. Thus, compared to the use of a loading process scheduled along with other processes, the load will not be spread along a time interval matching the scheduler resolution.

The main concern of such a solution is that it can significantly alter the scheduling of other processes in the system depending on the resolution at which the loading process generates its activity. If the resolution is too coarse, the overlapping of computation with I/Os might be hindered by large time intervals of CPU unavailability resulting from the load generation. Unfortunately, a fine resolution is not achievable because of the lack of precision in the scheduling control: to give the CPU back to users the loading process has to yield itself, activating the scheduler by doing so. This is done by putting the real time process into sleep. Nevertheless, due to OS implementation tricks, sleeping during a very short time interval does not necessarily imply a context switch to another process: sleep intervals have to be *large*. All these drawbacks have been outlined in the article that presents the current Wrekavoc release [15]. In Section IV-C we confirm these conclusions.

3) Solutions based on CPU frequency scaling: Nowadays, most processors provide a mechanism to dynamically downscale their operating frequency. This mechanism has been created as a way to reduce CPU power consumption when it is idle or moderately loaded: at a lower operating frequency, the CPU can also be operated at a lower voltage. Knowing that a CPU power consumption depends linearly on its frequency and quadratically on its voltage, this can lead to significant gains. Applications can easily make use of this feature using, for instance, the Linux `cpufreq` driver.

Besides its intended use, this mechanism can also be viewed as a way to reduce CPU performance (see [16] for its use as CPU performance degradation tool in a network performance study). This CPU performance degradation capability can also be viewed as a CPU load simulation. Nevertheless, several limitations of this mechanism prevent it from being an ideal solution for load generation. The first limitation comes from the implementation of this mechanism: it is not possible to downscale the CPU frequency to any chosen value, as only a limited set of possible choices are made available by the CPU maker. Consequently, it is not possible to generate any desired load profile using this mechanism. The second limitation is related to CPU models: few of them can downscale independently the frequency of distinct CPU cores to different values. These two limitations make this solution unsuited to the generation of realistic load profiles on many-core systems. They also make the solution much less versatile than KRASH.

Finally, one might wonder if reducing the CPU frequency is equivalent to make the CPU unavailable during some timeslices according to a desired load profile. Indeed such frequency scaling does not preserve the frequency ratio between the CPUs and the memory elements of the system. Most likely, the resulting tasks scheduling and overlapping between computations and I/Os will be different. This last issue will be investigated in more details in Section IV-C.

4) Qualitative comparison summary: In summary, the main issue with existing solutions is their inability to precisely reproduce any dynamic load profile. Beside this missing feature, we guess that some of these methods will raise obtrusiveness and realism issues. This last claim will be confirmed in Section IV-C. Table I presents an overview of our qualitative analysis.

C. Quantitative Comparison with Other Solutions

In this section, we compare KRASH to existing solutions from a pure performance point of view. We evaluate the performance of each tool by looking at its precision, its obtrusiveness and its intrusiveness. As outlined in Section IV-B, none of the other solutions we know about is able to generate an arbitrary dynamic load profile. Thus, we will compare KRASH with them regarding only constant load generation.

The main drawback of using constant load generation is that we cannot evaluate its result by using the generated profile: all the methods we compare have enough resolution to produce a constant perceived load. Yet, we can still measure the precision of each method: the run applications slowdown directly depends on the amount of generated CPU load. We can also evaluate the generator obtrusiveness: if an additional load is produced on other resources (such a I/Os) as a side effect, this will affect the slowdown of applications that depend on these resources. In both cases, reporting the concurrent application execution time will enable us to check the correct behavior of the generation method. Regarding the generator intrusiveness, we can measure it directly as the load induced by the supervisor process (if any).

1) Precision and Intrusiveness Comparison: In this experiment, the application we use is an instance of NAS NPB EP on 8 CPU cores. EP is a pure CPU intensive benchmark, thus whatever the load generation method, perturbations in other parts of the system (obtrusiveness) should not be noticeable. The goal of this experiment is to evaluate the precision, the resolution and the intrusiveness of each load generation method. We have measured the execution time of the NAS NBP EP benchmark on an unloaded machine as reference time. Then, for each method, we have checked that the generated load profile is near constant and we have measured the execution time of NAS NBP EP when all the cores are loaded at 50%. We deduce from the application slowdown the load it perceives in each case. Of course, in this pure CPU-related experiment, the perceived load should be as close as possible to 50%. When the load generation method involves a supervisor process, we also reported the load induced by this process (intrusiveness). The results of this experiment appear in Table II.

In these results, the resolution we report is an estimation based on the implementation chosen for each method.

	General dynamic load profile	Side effects on scheduler	Arbitrary number of processes	Intrusiveness	Resolution
KRASH [11]	Yes	Negligible	Yes	Negligible	Same as scheduler
Wrekavoc [15]	Not implemented	Low	No	Active poll	Higher than scheduler
Real time priority [5]	Not implemented	High	Yes	Periodic wakeup	Poor
Frequency scaling [16]	No	Medium	Yes	Negligible	Higher than scheduler

Table I: Features comparison table for several load generation solutions.

	NAS EP execution time		Perceived load		Intrusiveness	Estimated resolution
	average	standard deviation	average	standard deviation		
None	21.4	0.1	NA	NA	NA	NA
KRASH	41.9	0.9	48.9	1.1	1% on one core	1
Wrekavoc	47.6	4.5	55	3.9	15% per user process	2
Real time priority	47.1	3.8	54.5	3.2	NA	1
Frequency scaling	45.3	0.3	52.7	0.5	NA	NA

Table II: Precision comparison table for several load generation solutions.

Wrekavoc can only stop processes during whole seconds. If we add the management time of the tool itself to this minimal inactivity period, we obtain a resolution which should be closer to 2 seconds rather than 1. Real time scheduling issues microsleep calls to lower its resolution and ensure the scheduler is activated several times each second. Thus, in the general case it should be able to reach the same resolution as the scheduler, 1 second. KRASH completely relies on the `group scheduling` feature to regulate the load: the load balance between boulders and other processes is performed by the scheduler itself. Consequently, its resolution is the same as the scheduler, 1 second. Finally, the resolution metric is not relevant to CPU frequency scaling method. This is because this method does not generate any observable load, although, as all the other methods, it induces a degradation in NAS execution time.

On this CPU-only intensive computation, the degradation should be as close as possible to 100%: twice the time to complete, which corresponds to a perceived load of 50%. KRASH, along with CPU frequency scaling, produces the best result: KRASH is better in average but CPU frequency scaling has a lower standard deviation. We guess that the worse results observed for Wrekavoc and Real time priority are due to their action from outside the scheduler: timeslices are divided less accurately between processes.

2) Obtrusiveness comparison: The obtrusiveness of a CPU-only load generation method encompasses all the side effects caused by the generation method on other resources in the system. System tasks that are the most affected by the kind of CPU load we generate are the ones that can usually be overlapped with computation tasks: networking operations and I/Os transactions. When hindering the scheduler efforts to nicely order running processes, a load generation method might prevent overlapping and induce worse than expected network or I/Os performance degradation. Consequently, we present in this section a collection of experiments that outlines the general side effects produced by each generation method on these resources.

The first experiment focuses on I/Os performance. It consists in running the command `dd` to copy a file on the local filesystem in a loaded machine. In this experiment the system write cache has no importance for two reasons. The first reason is that we give the `fdatasync` option to `dd` which forces physical write at the end of the copy. The second reason is that the most important performance factor in this experiment is the scheduling of I/O tasks. These tasks issue write operations with a system call, posting a command in the write queue of the I/O scheduler. They often need to be scheduled as soon as possible to benefit from overlapping with computation. This claim is validated by the fact that we obtain the same results without the `fdatasync` option. Table III presents the results of this experiment when copying a 1GB file on a machine loaded at 50%.

	Time to copy the file		Slowdown
	average	standard deviation	
None	10.2	0.8	1
KRASH	20.5	0.5	2
Wrekavoc	24.9	1.7	2.4
Real time priority	36.6	1.8	3.6
Frequency scaling	24.3	1.8	2.4

Table III: Side effects on I/Os evaluated for several load generation solutions.

As we can notice, only KRASH produces the expected slowdown of 2. It is the only method that lets the scheduler use its usual optimizations: I/O tasks are scheduled in priority (as long as the remaining loading process can get the desired 50% of CPU time). The CPU frequency scaling solution lets the scheduler use these optimizations, but the degraded CPU frequency results in a coarser granularity and a higher than expected load. Wrekavoc and Real time priority produce even worse results. This is because they disturb the scheduler work: signals issued by Wrekavoc to regulate the `dd` load might occur just before blocking I/O operations, unscheduling the task twice: one time with the signal and the second time when performing the blocking call. Furthermore, the supervisor process polls the `/proc` virtual filesystem resulting in even more I/O operations. Regarding Real time priority, the change in the scheduler policy (FIFO scheduling) simply alters the usual I/O optimizations. It appears that this last strategy is the

worst one regarding I/Os obtrusion.

The next experiment focuses on the effects of our generation methods on network performance. The benchmark we use in our evaluation is NAS NBP DT, which performs intensive point-to-point blocking MPI communications between all the involved processes. We run this benchmark with 80 processes and a random communication topology. The slowdowns we obtain on this benchmark when loading the machine at 50% with our different methods are reported in Table IV.

	Execution time		Slowdown
	average	standard deviation	
None	2.9	0.5	1
KRASH	6.2	0.8	2.1
Wrekavoc	NA	NA	> 100
Real time priority	11.3	3.2	3.9
Frequency scaling	4.4	0.6	1.5

Table IV: Effects of load generation solutions on NAS NBP DT.

We can point out that CPU frequency scaling produces a moderate slowdown of 1.5 only, rather than 2 as we could have expected when loading the machine at 50%. This surprising result is explained by the way this application works: distinct MPI processes synchronize on their blocking point-to-point communication. Thus, the overlapping between computation and communications occurring in different processes plays a significant role. This slowdown of 1.5 just outlines the fact that the communications are not fully overlapped with computations when running the benchmark on an unloaded machine: there is still room for more overlapping on a slower CPU. In contrary, with KRASH, the CPU is made unavailable during long periods (whole timeslices). This does not increase the number of opportunities for further computation/communications overlapping. This is also what would a real concurrent application do. It results in an expected slowdown of 2. Real time priority degrades twice as desired the performance of this benchmark, for the same reasons as in the I/O case: blocking operations are not given priority because of the FIFO scheduling policy. Finally, in this experiment, Wrekavoc produces exceedingly high additional system load. The resulting execution time roughly corresponds to a slowdown of 100. This is because Wrekavoc creates a new supervisor process for each new user process: with 80 processes in our benchmark this becomes a serious bottleneck. This confirms the claim we made in Section IV-B that Wrekavoc is not suited to load more processes than available CPU cores.

The last benchmark is a compilation of `gcc` with a maximum of 16 parallel processes. This benchmark involves processes with varying duration executing a mix of CPU intensive and I/O operations (compilation of more than 30000 files with a total size of 300MB). As in previous experiments, we inflict a load of 50% on all the CPU cores. The resulting slowdown is reported in Table V.

In this benchmark, the pure I/Os performance is less critical that in the case of `dd`. Thus, CPU frequency scaling and KRASH perform as well, slowing down the compilation by an expected factor of 2. For the same reasons as in the case of `dd`, Real time priority degrades more than desired the execution, resulting in a slowdown of 2.8. Finally, Wrekavoc exhibits the same behavior as in the case of the NAS NBP DT benchmark: it generates an exceedingly high system overhead, although, in this case, there are never more running processes than CPU cores. The reason is that the compilation of gcc involves the execution of many low duration processes. In this case, the overhead of the supervisor processes used by Wrekavoc (creation, `/proc` scanning, and termination) consumes most of system resources.

	Execution time		Slowdown
	average	standard deviation	
None	197	3	1
KRASH	387	7	2
Wrekavoc	NA	NA	> 100
Real time priority	558	5	2.8
Frequency scaling	392	21	2

Table V: Effects of load generation solutions on gcc compilation.

V. (Un)Related Works

The load injection research area focuses on producing stressing situations to test various kinds of server-type computing systems (I/O, web, routing). CLIF [8] is an example of well known load injection tool. Despite the similarity of our intention (producing a load), the topic we address in this article is radically different from load injection concerns. In the context of load injection, the system is viewed as a black box on which we act by fetching input data that induce computation. The main issue in this context is to determine statistical distributions of input data scenarii and their effect on the observed system. In this article we are not interested in statistical characterization of load profiles, but rather in system mechanics that enable us to reproduce a desired CPU load. Thus, this article is not to be misinterpreted as an article about load injection.

Benchmarking consists in executing a program written specifically to test the performance of a given system. The intent of this test is either to evaluate the overall performance when executing an application that belongs to a given class (NAS NPB [7], LINPACK [9], HPCC [10]) or to evaluate the performance of a specific part of the executing platform (HBench:OS [3], SPEC MPI [14]). In benchmarking, the goal is to produce a focused intensive activity on an unloaded system to evaluate its performance. In this article we aim at making the CPU unavailable according to a given load profile in a reproducible way. The activity of the CPU during this unavailability periods (intensive activity or simple empty loop) does not matter. Thus, results from the benchmarking research area cannot be of any help to solve our problem.

VI. Conclusion

In this article, we have presented a new method for the reproducible generation of CPU load in a shared memory machine. As we outlined in this whole work, the main issue in this context is reproducibility whatever the present environment

on the target machine (other processes, allocated resources). Aiming at ideal reproducibility induces other objectives as a side effect: precision, unobtrusiveness and, most of the time, unintrusiveness. We presented a new methodology for CPU load generation that cooperates with the system scheduler to reach our objectives. We argue that this method is the only way to generate a dynamic load profile with the best precision. We also presented an implementation of this method for Linux taking advantage of recent kernel capabilities.

We compared our implementation to other CPU load generation tools and methods. As we emphasized in our qualitative comparison, our tool KRASH is the only one that is able to generate any dynamic load profile, while the others are usually restricted to continuous loads. Then, our quantitative comparison showed that some of the existing methods have noticeable intrusiveness issues under high system load conditions. Furthermore, because they act from outside the scheduler, all these methods have serious obtrusiveness issues. In some case this even makes the whole generator meaningless and unusable. Because of its conception, our method avoids these pitfalls and always precisely generates the desired load without noticeable intrusiveness or obtrusiveness. KRASH is available for download at http://krash.ligforge.imag.fr, it is provided free of charge under the terms of the GNU GPL license.

KRASH is expected to be used as an experimentation environment for adaptive parallel algorithms efficiency comparison. It is able to produce dynamic CPU loads on shared memory machines that are similar to real loads. While the CPU is usually the most important resource in high performance computing, some applications also heavily depend on other parts of the system such as caches, memory, I/O subsystem or network. Our future works will focus on extending KRASH to make it able to generate load on these other parts.

REFERENCES

[1] Robert D. Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Cambridge, MA, USA, 1995.

[2] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymont Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Journal of High Performance Computing*, 4(20):481–494, 2006.

[3] Aaron Baeten Brown. A decompositional approach to computer system performance evaluation, October 03 1997.

[4] Albert Cahalan. PROCPS, 1997-2008.

[5] Louis-Claude Canon and Emmanuel Jeannot. Wrekavoc: a tool for emulating heterogeneity. In *15th IEEE Heterogeneous Computing Workshop (HCW 06)*, 2006.

[6] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: a generic framework for large-scale distributed experiments, April 14 2008.

[7] David Bailey et al. The NAS parallel benchmarks. Technical Report RNR-91-002, NAS Systems Division, January 1991.

[8] Bruno Dillenseger. Flexible, easy and powerful load injection with clif version 1.1. In *In Fifth Annual ObjectWeb Conference*, 2006.

[9] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, August 2003.

[10] Jack Dongarra, Piotr Luszczek, and Allan Snavely. HPC challenge (HPCC) benchmark suite. In *SC'07 USB Key*. ACM/IEEE, Reno, NV, November 2007.

[11] KRASH: Kernel for Reproduction and Analysis of System Heterogeneity. http://krash.ligforge.imag.fr/, 2008-2009.

[12] Sebastien Godard. SYSSTAT, 2002-2008.

[13] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *JPDC: Journal of Parallel and Distributed Computing*, 63, 2003.

[14] Matthias S. Muller, Kumaran Kalyanasundaram, Greg Gaertner, Wesley Jones, Rudolf Eigenmann, Ron Lieberman, Matthijs Van Waveren, and Brian Whitney. SPEC HPG benchmarks for high-performance systems. *Int. J. of High Performance Computing and Networking*, 1:162–170, December 07 2005.

[15] Jens Gustedt Olivier Dubuisson and Emmanuel Jeannot. Validating Wrekavoc: a tool for heterogeneity emulation. In *18th IEEE Heterogeneous Computing Workshop (HCW 09)*, 2009.

[16] Ravi Iyer Srihari Makineni. Measurement-based analysis of tcp/ip processing requirements. In *In 10th International Conference on High Performance Computing (HiPC 2003)*, 2003.

[17] Daouda Traore, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. Adaptive parallel algorithms and applications to stl. In Springer-Verlag, editor, *EUROPAR 2008*, Las Palmas, Spain, August 2008.