HAL
open science

GenPack: A Generational Scheduler for Cloud Data Centers

Aurélien Havet, Valerio Schiavoni, Pascal Felber, Maxime Colmant, Romain Rouvoy, Chistof Fetzer

To cite this version:

Aurélien Havet, Valerio Schiavoni, Pascal Felber, Maxime Colmant, Romain Rouvoy, et al.. GenPack: A Generational Scheduler for Cloud Data Centers. 5th IEEE International Conference on Cloud Engineering (IC2E), Apr 2017, Vancouver, Canada. pp.10. hal-01403486

HAL Id: hal-01403486

https://inria.hal.science/hal-01403486v1

Submitted on 13 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GENPACK: A Generational Scheduler for Cloud Data Centers

Aurelien Havet*, Valerio Schiavoni*, Pascal Felber*, Maxime Colmant†, Romain Rouvoy†, Christof Fetzer‡
*University of Neuchâtel, Switzerland. Email: first.last@unine.ch
†ADEME / Univ. Lille / Inria, France. Email: maxime.colmant@inria.fr
‡Univ. Lille / Inria / IUF, France. Email: romain.rouvoy@inria.fr
§Technical University of Dresden, Germany. Email: christof.fetzer@tu-dresden.de

***Abstract*—Cloud data centers largely rely on virtualization to provision resources and host services across their infrastructure. The scheduling problem has been widely studied and is well understood when the resource requirements and the expected lifetime of services are known beforehand. In contrast, when workloads are not known in advance, effective scheduling of services, and more generally system containers, becomes much more complex. In this paper, we propose GENPACK, a framework for system containers scheduling in cloud data centers that leverages principles from generational *garbage collection* (GC). It combines runtime monitoring of system containers to learn their requirements and properties, and a scheduler that manages different generations of servers. The population of these generations may vary over time depending on the global load, hence they are subject to being shut down when idle to save energy. We implemented GENPACK and tested it in a dedicated data center, showing that it can be up to 23% more energy-efficient that SWARM's built-in scheduling policies on a real-world trace.**

***Keywords*-generational scheduling; docker; energy efficiency;**

I. INTRODUCTION

Resource allocation in cloud data centers is an important yet complicated problem. On the one hand, over-provisioning tends to waste resources—be they monetary or environmental. On the other hand, overbooking yields poor performance and may lead to *service level agreement* (SLA) violations, which also has financial consequences.

To increase the flexibility of task management, cloud data centers largely rely on virtualization to run applications and services for their customers. While some providers offer dedicated servers at a premium price, most usually they co-locate several services and/or jobs on the same physical servers in order to optimize the use of available resources and reduce the associated costs.

Efficient mapping of services and jobs—packaged as system containers—to hosts is non-trivial as it should take into account, not only the resources available on the possibly heterogeneous machines, but also the properties and requirements of the containers. For instance, some containers might require much memory but little CPU or I/O resources, while others are CPU-intensive, or primarily perform network and disk accesses. To make the problem worse, these properties and requirements are not necessarily known in advance and must be learned at runtime.

In this paper, we therefore introduce GENPACK, a novel scheduling framework for containers placement and migration in cloud data centers, which leverages principles from generational *garbage collection* (GC) [1], [2]. The core idea of GENPACK is to partition the servers into several groups, named *generations*. A first generation of servers, the *nursery* generation, hosts new containers whose workload are not known. There, the system containers are automatically monitored to determine their resource profile on reference machines in order to learn their characteristics. To that end, we designed a monitoring framework that combines local statistics and power estimations from the CADVISOR [3] and BITWATTS [4] agents.

Once their workload is properly understood, the system containers are migrated to a server of the second generation, the *young* generation. The placement of system containers in the *young* generation is performed according to resource-aware scheduling policies, and the servers in this generation are in charge of hosting containers whose lifetime is relatively short or unknown.

Finally, if a container runs for long enough in the young generation, it will be migrated to the *old* generation. Servers in this last generation are the most stable and tend to host long-term containers. Placement is performed so as to optimize the load of the machines by co-locating containers that have complementary resource requirements. For instance, a node that has high CPU utilization, but underloaded memory, will be candidate to host a memory-intensive container with low CPU requirements.

GENPACK allows us to take advantage of the different properties of the server generations and the system containers that they host to flexibly provision resources and thus save energy. New machines can be added to each generation as needed.This allows us to elastically adapt to demand and load variations—*e.g.*, between day and night—and take advantage of server-specific properties—*e.g.*, use the most energy-efficient machines for the old generation. Furthermore, by rationalizing the usage of some servers while shutting down others, one can reach closer to *energy-proportional computing* [5].

GENPACK provides several key original features: it supports heterogeneous data centers and servers with different properties (*e.g.*, single- vs. multi-core, energy-efficient vs.

fast, with or without HW acceleration, etc.); it supports containers whose workload and duration are not known in advance (which is the general case for many application domains) and must be learned at runtime; it supports fluctuating workloads by adapting the number of servers in the different generations, thus enabling energy-efficient container scheduling in cloud data centers.

We have implemented our approach within the DOCKER SWARM framework [6]. In particular, GENPACK includes a comprehensive monitoring framework, as well as resource management, container migration, and scheduling mechanisms. We have tested our system in a dedicated data center with real-world traces from [7]. Our evaluation reveals that GENPACK is up to 23% more energy-efficient than SWARM's built-in schedulers with a real-world trace.

This paper is organized as follows. We first introduce a motivating scenario in §II and describe the overall architecture of GENPACK in §III. We present the monitoring framework and the scheduling mechanisms respectively in §IV and §V. We briefly discuss some implementation notes in §VI and provide a comprehensive evaluation in §VII. Finally, we review related work in §VIII and conclude in §IX.

II. MOTIVATING SCENARIO

To illustrate and assess the benefits of proper container (or VM)[1] placement, we first illustrate the limitation of existing scheduling policies on a simple scenario.

We define two types of containers: *cpu-heavy* containers require 2 CPU cores and 1 GB of RAM, while *mem-heavy* containers require only 1 CPU core but 2 GB of RAM. We set up a cluster of nodes with 8 available cores and 8 GB of RAM, running UBUNTU SERVER (v15.10) and DOCKER (v1.10.1). The containers are managed by Docker Swarm (v1.2.0) and they execute the STRESS-NG benchmark [8] with a fixed total number of operations before terminating.

We deploy the containers in a dedicated cluster using four placement strategies:

- `spread` places new containers on the node with the least number of containers;
- `binpack` deploys containers on the same node until its resources are totally exhausted before moving to the next node;
- `random` dispatches containers at random;
- `custom` assigns containers to nodes so that they fit into the least number of nodes, by taking into account both the CPU and memory requirements.

For the sake of illustration, assume that a node can host *(i)* 3 *cpu-heavy*, or *(ii)* 3 *mem-heavy*, or *(iii)* 2 *cpu-heavy* and 2 *mem-heavy* containers of each type. In that case, a scheduler that takes into account the nature of the workload can obviously perform more efficient container placement.

[1]In the remaining of the paper, we primarily consider containers, which are essentially lightweight VMs, and we use the two terms interchangeably.

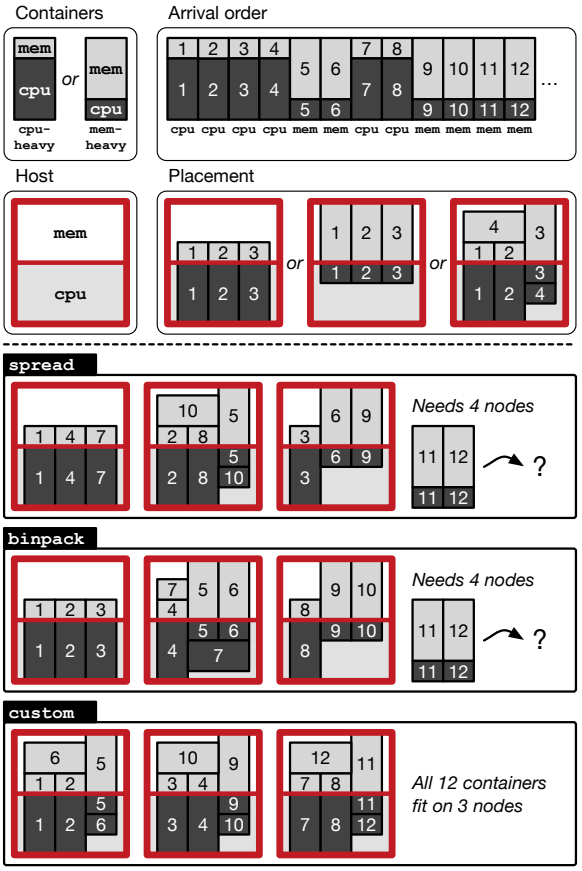

Figure 1. Placement of the containers with 3 scheduling strategies for a given arrival order of containers, and assuming that a node can host 3 *cpu-heavy* containers, or 3 *mem-heavy* containers, or 2 of each type (top). While `spread` and `binpack` would require 4 nodes to schedule 12 containers, `custom` requires only 3 (bottom).

Figure 1 shows a simple execution where the 12 containers (6 of each type) are registered in the following order: 4 *cpu-heavy*, 2 *mem-heavy*, 2 *cpu-heavy*, 4 *mem-heavy*. Containers specify their resource needs and the system performs placement accordingly without overbooking. A possible container scheduling for the `spread`, `binpack`, and `custom` strategies is shown in the bottom part of the figure. As one can see, with 3 nodes available the first two strategies can only schedule 10 containers, whereas the `custom` strategy can place all of them on the 3 nodes. Although very simplistic, this example illustrates the need for scheduling strategies that are aware of the requirements of the containers and the properties of the workloads.

In our actual experiment, we set the CPU load of containers to 20,000 "bogo" operations[2] for each CPU core. This corresponds to a total of $40,000$ and $20,000$ operations for *cpu-heavy* and *mem-heavy* containers, respectively. Figure 2

[2]Fake operations that represent the unit of load of the benchmark.

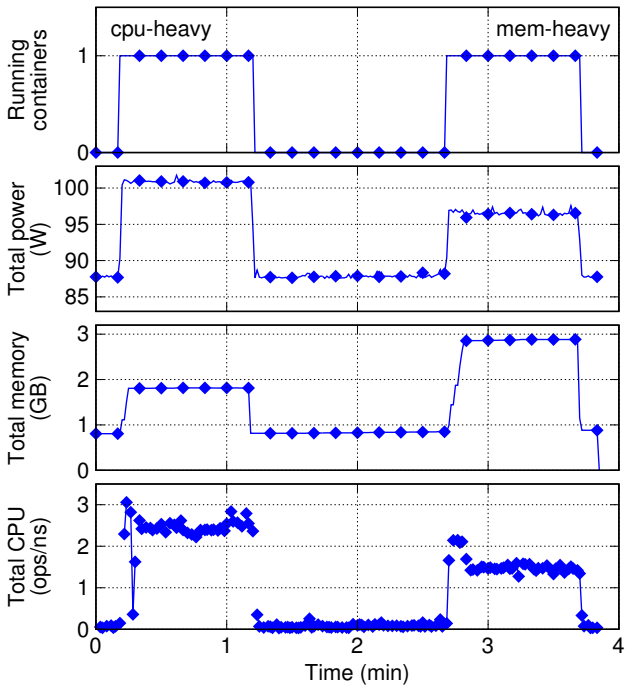

Figure 2. Workload for the two container types (*cpu-heavy* and *mem-heavy*) deployed on a single host. Each container runs for one minute, with an idle period in between.

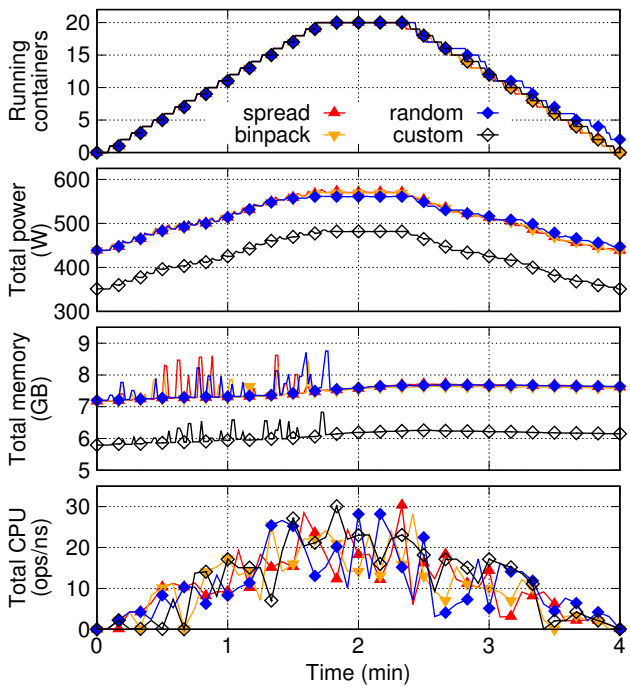

Figure 3. The `custom` strategy saves memory and energy without affecting performance thanks to more efficient container scheduling.

shows the baseline workloads induced by the two types of containers deployed on a single node, running one after the other within the span of 5 minutes. As expected, *cpu-heavy* containers consume more energy—if we subtract the idle power, they require almost 50% more than *mem-heavy* containers—but they are less memory demanding.

Then, we design a more elaborated deployment scenario where we deploy start 20 containers, alternating 5 *cpu-heavy* and 5 *mem-heavy*. In all four deployment scenarios, we gather several measures (*e.g.*, memory allocations, CPU usage, power consumptions). Results are aggregated values over all nodes: number of CPU operations by time unit (ns), memory used in GB, and cumulative power (idle power and dynamic consumption) in W. We observe that the `custom` strategy results in a more memory- and energy-efficient schedule because one of the nodes can be turned off—hence saving the idle power—without noticeably changing the number of operation executed—*i.e.*, performance.

As a summary, we aim at delivering a new container scheduler, GENPACK, that automatically learns from container's workloads to evenly distribute their deployment across a reduced number of nodes, thus drastically improving the power usage efficiency of a cluster. As demonstrated, the state-of-the-art fails to achieve this objective as the `spread` strategy distributes the containers across all the nodes and `binpack` adopts a greedy heuristics to allocate containers node per node. Given a set of available nodes N, we therefore aim at proposing a solution that minimizes the number of powered nodes required to host a set of containers C, thus ensuring that $|\text{genpack}(N,C)| \leq |\text{binpack}(N,C)| \leq |\text{spread}(N,C)| = |N|$, where $\text{f}(N,C)|$ indicates the number of nodes necessary for scheduling C containers on N nodes using algorithm f. By doing so, GENPACK reduces the overall power consumption of a cluster without impacting the containers' performance, since hosts are not energy proportional (as shown in Figure 2).

III. GENPACK ARCHITECTURE

In this section, we provide an overview of the software architecture of GENPACK, as well as its main components and their interactions. Further details about the implementation of these components and how they operate are given in the following sections.

A. Generations

As previously mentioned, GENPACK splits the nodes of a cluster into multiple generations, each responsible for specific types of containers and tasks. The rationale is that, by specializing nodes for a given type of workloads, one can handle system containers whose properties are not known in advance or are dynamically evolving, while at the same time optimize the whole system's efficiency by placing the containers on the most appropriate nodes.

Along the same lines as generational garbage collectors in managed languages, GENPACK considers 3 generations: the nursery, the young generation, and the old generation, as illustrated in Figure 4.

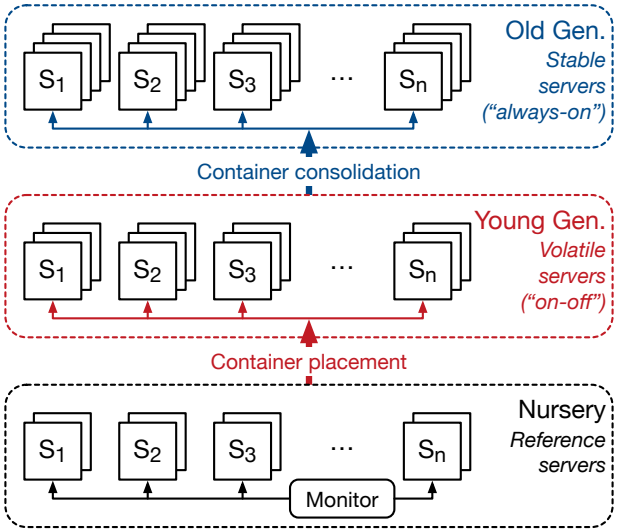

Figure 4. GENPACK's different generations.

The *nursery* consists of a set of reference nodes that are representative of servers in the data center and whose properties are well understood. A container that has not yet been profiled will first execute in the nursery.[3] During its first period of execution, GENPACK will monitor its resource requirements as well as its power consumption. To that end, it leverages system-level metrics provided by CADVISOR (see §IV) and power information from BITWATTS [4]. This observation phase allows GENPACK to establish a profile for the container.

Once a container's properties are known, and assuming that it did not complete its execution, it is moved to the *young generation* ("placement" phase) on a server that has sufficient resources available considering the container's specific requirements (CPU, memory, network, etc.). The young generation hosts containers that have recently started their execution and whose lifetime is still unknown. If the container survives long enough, it moves to the next generation. The reasoning behind this placement strategy is that, similarly to in-memory data objects, a significant portion of the containers are expected to have a short lifetime.[4]Furthermore, as the young generation is the most exposed to load variations (*e.g.*, when many new containers are simultaneously launched), it will provide mechanisms for elastically scaling up and down, according to demand. In particular, nodes can be completely turned off during periods of low load in order to considerably reduce the energy consumption of the cluster.

Finally, the *old generation* consists of stable and power-efficient servers that host the long-running containers. The placement of containers on the nodes ("consolidation" phase) is performed in such a way that they occupy the minimum number of servers in order to optimize resource and power usage, as motivated in §II. Barring important workload variations, containers do not need to migrate further once they are on an old generation node.

[3]Note that containers can skip the nursery and directly go to the next generation when previously profiled.

[4]We assess this statement in Section VII.

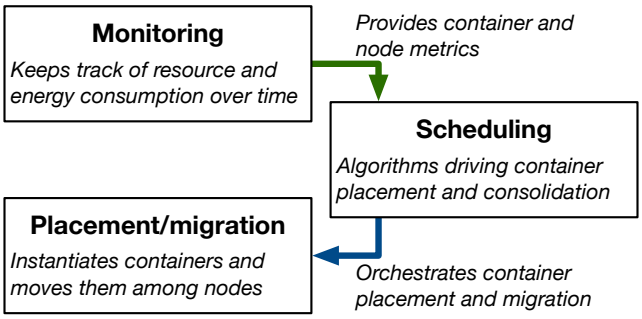

Figure 5. GENPACK's abstract components and their interactions.

The actual monitoring and scheduling operations that drive the migrations between generations are described in §IV and §V.

B. System components

From a high-level perspective, GENPACK is composed of three main components:

- The *monitoring* module is responsible for keeping track of resources consumption in the system.
- The *placement and migration* module handles the deployment of containers and their relocation to different nodes as they move across generations.
- The *scheduling* module contains the algorithms that orchestrate and take decisions regarding container placement and migrations, based on the input received from the monitoring module.

The role and interactions between these components are schematically illustrated in Figure 5.

IV. CONTAINER AND NODE PROFILING

System containers can exhibit a wide diversity of properties and requirements, from CPU-intensive tasks running for a short duration to longstanding memory-intensive applications serving user requests. In such a context where the workloads are unknown, it is particularly challenging to ensure an efficient scheduling of these containers. We therefore propose to introduce a resource profiling phase within GENPACK to automatically learn the resource requirements of a container during the beginning of its execution, and to subsequently use this information to compute a resource envelope that will help the GENPACK scheduler to appropriately place the container on the best fitting node for the rest of its execution. In particular, this container profiling phase is performed within the *nursery* and *young* generations of GENPACK and is complemented with a monitoring of the nodes located in the *young* and *old* generations in order to

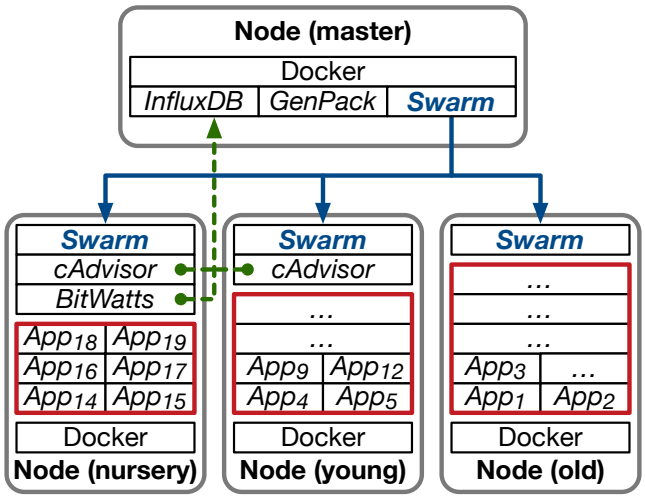

Figure 6. Overview of the monitoring support in GENPACK.

maintain a up-to-date cartography of available resources in the cloud data center.

Profiling the resources consumption.: Upon deployment of a new container within the *nursery* generation, GENPACK uses a CADVISOR daemon [3] to collect, aggregate, process, and export metrics about running containers every 30 seconds. In particular, CADVISOR logs resource isolation parameters, historical resource usage, histograms of complete historical resource usage, and network statistics for each system container running on a DOCKER host. Collected metrics are automatically exported towards an INFLUXDB service [9] hosted on the master node (see Figure 6). INFLUXDB provides a time-series database to store cluster-wide metrics per container, according to a specific data retention policy (x minutes in GENPACK). Whenever needed, GENPACK can therefore query INFLUXDB to learn about the containers' workloads.

Computing the container envelopes.: Periodically, GENPACK picks the containers running in the *nursery* generation and triggers a scheduling phase for all of them. As part of this phase, GENPACK queries INFLUXDB to convert raw resource metrics into *container envelopes*, which will be used by the scheduler to estimate the expected resource consumption. In particular, for each resource, GENPACK first computes the metrics distribution and extracts the 90th *percentile* value as a component of the resource envelope. Then, GENPACK splits the set of containers into k clusters by applying the *k-means* algorithm, which belongs to the category of unsupervised learning approaches. For example, we can set $k = 4$ to segregate 4 classes of CPU-, disk-, network-, and memory-intensive workloads into 4 container envelopes.

Finally, within each envelope, containers are ordered per decreasing resource consumption score, which is computed

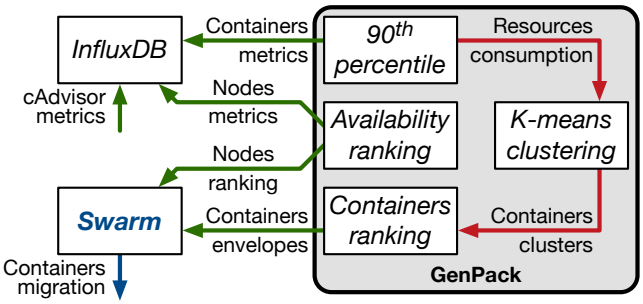

Figure 7. Container and node profiling in GENPACK.

for each enclosed container i as:

$$score_i = \sqrt{(\frac{cpu_i}{\sum cpu})^2 + (\frac{disk_i}{\sum disk})^2 + (\frac{net_i}{\sum net})^2 + (\frac{mem_i}{\sum mem})^2}\,.$$

The resulting container envelopes are posted to the GENPACK scheduler, which is in charge of placing the containers among the nodes of the *young* generation.

Beyond this first scheduling phase, GENPACK keeps monitoring and profiling the containers within the *young* generation in order to consolidate the resource envelope prior to a later migration in the *old generation*.

Maintaining the node availability cartography.: GENPACK monitors the resource availability of nodes within the *young* and *old* generations. For each generation, it uses this information to rank the nodes according to resource availability, least available nodes first, by computing for each node j the availability level as:

$$availability_j = \sqrt{cpu^2_{ratio} + disk^2_{ratio} + net^2_{ratio} + mem^2_{ratio}}\,,$$

which corresponds to the norm of the resource vector $\vec{r_j} = (cpu_{ratio} \quad disk_{ratio} \quad net_{ratio} \quad mem_{ratio})$ that GENPACK extracts from INFLUXDB. This ranking of nodes will then be used by the scheduler to find the first fitting node to host a container, ultimately minimizing the number of hosts to be used—*i.e.*, that need to be powered up.

V. CONTAINER SCHEDULING

Once the container profiles are identified and the associated resource envelopes have been computed by the monitoring module of GENPACK, the scheduling module builds on these resources estimations to identify the best fitting node for each of the container executing in the *nursery* generation.

More specifically, Algorithm 1 describes the scheduling strategy applied by GENPACK to migrate a set of profiled containers at runtime. The scheduling phase is triggered for a given set of container $envelopes$ and available $nodes$. The algorithm starts by homogeneously blending the content (*i.e.*, container descriptions) of the envelopes (line 2 and lines 17–31) to increase of the diversity of containers per node. From there, it iterates over this ordered set of $containers$ to be scheduled (line 5) and picks the first node n among the ordered list of available $nodes$ (as

Algorithm 1 Container scheduling in GENPACK.

```
1: procedure SCHEDULE(envelopes, nodes)
2:     containers ← BLEND(envelopes)
3:     for c ∈ containers do
4:         res_c ← RESOURCES(c)
5:         for n ∈ nodes do        ▷ Find the best node for c
6:             avail_n ← AVAILABILITY(n)
7:             if MATCHES(res_c, avail_n) then
8:                 n ← UPDATE(n, res_c)
9:                 nodes ← SHIFTLEFT(nodes, n)
10:                MIGRATE(c, n)        ▷ Async. migration
11:                break        ▷ c succeeds to be scheduled
12:            end if
13:        end for
14:        ESCAPE(c)        ▷ c fails to be scheduled
15:    end for
16: end procedure
17: function BLEND(envelopes)
18:    list ← {}
19:    emptied ← true
20:    repeat        ▷ Blend until all envelopes are emptied
21:        emptied ← true
22:        for env ∈ envelopes do
23:            if not ISEMPTY(env) then
24:                list ← list ‖ HEAD(env)
25:                env ← TAIL(env)
26:                emptied ← ISEMPTY(env)
27:            end if
28:        end for
29:    until emptied
30:    return list
31: end function
32: function SHIFTLEFT(nodes, node)
33:    i ← INDEX(nodes, node)
34:    n ← LENGTH(nodes)
35:    list ← nodes[0 : i − 1] ‖ nodes[i + 1 : n − 1]
36:    i ← 0
37:    score ← AVAILABILITY(node)
38:    while score ≥ AVAILABILITY(list[i]) do
39:        i ← i + 1
40:    end while
41:    return list[0 : i − 1] ‖ node ‖ list[i : n − 1]
42: end function
```

explained in the previous section) that matches the resources requirements of the container c (line 7). Upon resource matching, the estimation of the node's resource availability is updated accordingly (line 8) and the order of available $nodes$ is refreshed by shifting the selected node n towards the head of the ordered list (line 9 and lines 32–42). By reasoning on such resources estimations, GENPACK can therefore trigger the migration of the container c to the node n asynchronously (line 10) and thus keep scheduling the remaining nodes in parallel. If none of the available $nodes$ fits the resource requirements of the container c, the escape trigger of GENPACK (line 14) is used to provision a new node within the *young* generation, migrate the container c on this new node, and add the node to the list of available nodes.

The intuition behind this algorithm is to ensure a better distribution of resource consumption and power efficiency of the infrastructure by increasing the entropy (in term of resource diversity) of the containers deployed within each node. Furthermore, by reasoning on resource estimations (computed during the monitoring phase) instead of real-time metrics, GENPACK can increase the scheduling parallelism and thus absorb the delay induced by the container migration process (including state snapshotting, binary transfer, remote provisioning steps).

VI. IMPLEMENTATION

DOCKER SWARM is implemented in Go, but it offers multiple bindings for other programming languages. We fully implement GENPACK in the Ruby programming language (v2.3.1). In particular, we based our code on gem *docker-api* [10], a lightweight Ruby binding for the DOCKER REMOTE API, using DOCKER API (v1.16) [11], compliant with the one used by DOCKER SWARM(v1.22). The scheduler orchestrates the containers by leveraging RESQUE [12], a REDIS-backed Ruby library for creating background jobs.

In our evaluation, we map containers to jobs and rely on a RESQUE-based scheduler to timely deploy them over the DOCKER SWARM. GENPACK is released as open-source and is freely available at https://bitbucket.org/GenPackTeam/genpack-testbed.

VII. EVALUATION

This section reports on a detailed evaluation of the GENPACK prototype. More specifically, we describe our experimental settings in §VII-A. §VII-B characterizes the real-word trace used in our experiments, focusing in particular on the simplifications that we applied to make it applicable in our dedicated data center. We present the performances of the GENPACK approach in terms of job completion and energy impact when compared against different default strategies in §VII-D and §VII-E, respectively. Finally, §VII-C offers an insider-view on the dynamics of the generations in terms of running containers.

A. Evaluation settings

We deploy and conduct our experiments over a cluster machines interconnected by a $1\,Gb/s$ switched network. Each physical host features 8-Core Xeon CPUs and $8\,GB$ of RAM. We deploy dedicated *virtual machines* (VM) on top of the hosts. The KVM hypervisor, which controls the execution of the VM, is configured to expose the physical CPU to the guest VM and DOCKER container by mean

of the `host-passthrough` option to access optimized CPU instruction sets [13]. The VMs exploit the `virtio` module for better I/O performances. For the sake of cluster management simplicity, we deploy the DOCKER daemon (engine v1.12) on top of the VMs. Note however that we did not observe sensible performance differences when deploying the DOCKER containers on bare-metal.

The scheduling of the containers is orchestrated by DOCKER SWARM (v1.2.0), the default scheduling framework supported by DOCKER. SWARM comes with a set of predefined scheduling strategies: we compare GENPACK against each of those along several axes [14]. The same strategies are supported by the recently released DOCKER ENGINE (v1.12) or other VM/container schedulers (*e.g.*, OPENNEBULA) [15].

Our cluster is composed of 13 hosts: one acts as the SWARM master node, orchestrating the deployments, while the remaining worker nodes join the SWARM pool to execute the jobs. The cluster thus accounts for 96 cores and 96 GB of RAM in total. Unless specified otherwise, the 3 generations used by the GENPACK strategy are composed as follows: 5 SWARM nodes in the *nursery*, 4 in the *young* generation, and 3 in the *old* generation.

B. Google Borg Trace

To evaluate GENPACK under realistic settings, we use a subset of the Google Borg Trace [16], [17]. The trace provides detailed informations about the duration of the jobs and their demanded resources (CPU quotas, memory, etc.). Note that it is outside of the scope of this paper to provide a full characterization of the Google trace, as several ones exist already [18]. The original trace is unmanageable for anyone but major companies with huge clusters that have sufficient hardware resources to met the demands of all concurrent jobs. Therefore, to deploy the workload into our data center while, at the same time, retaining the same overall workload and realistic load patterns, we sample the original trace to deploy $1/100^{th}$ of the original jobs.

In terms of resource requirements, the original trace describes each job's demanded resources scaled to the Google's most powerful node in that given Borg cell (*e.g.*, a request for 1.0 would map all the CPU cores of a machine to a given job, and similarly for the memory). We follow the same principle by mapping those to the hardware resources available in our cluster.

Figure 8 shows the dynamics of the sampled workload in terms of concurrently executing jobs. The sampled trace consists of 49, 202 jobs, with a peak of 102 concurrent jobs, and an average job submission rate of 68.3 jobs per minute.

For practical reasons, our experiments only consider the first 12 hours of the trace, instead of the whole available period of 29 days. Jobs that cross the 12-hour mark are killed abruptly. We filter out jobs longer than 50 minutes, as they represent less than 20% of the jobs in the original

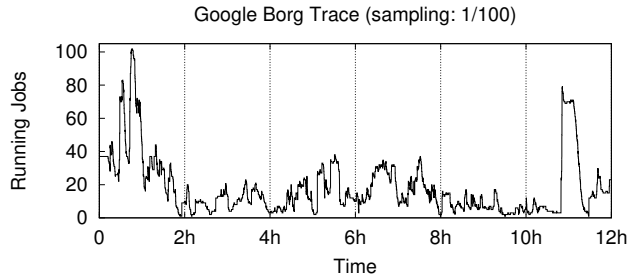

Figure 8. Initial 12 hours of the Google Borg Trace.

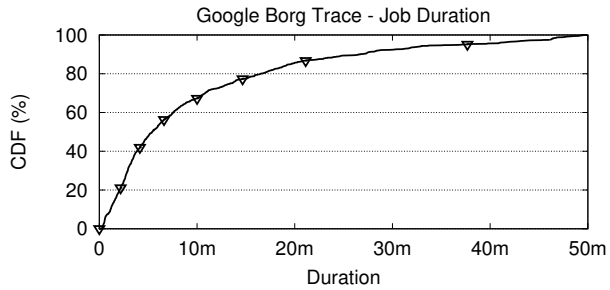

Figure 9. Google Borg Trace: CDF of job durations.

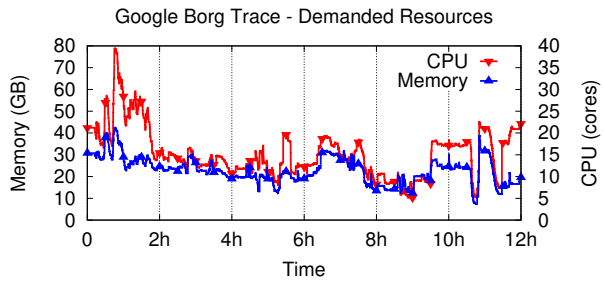

Figure 10. Google Borg Trace: demanded resources.

trace and are not very meaningful given the 12-hour period consider. Furthermore, in our sampling we only take into account the jobs that complete successfully.

Figure 9 presents the duration of the jobs from the sampled trace as a *cumulative distribution function* (CDF). The considered jobs have a lifespan between 39 s (the 5*th* percentile) and 50 m (the 100*th* percentile). Figure 10 depicts the memory and CPU workload injected by the trace on our cluster. We observe peak allocations of 42 cores and 40 GB of total memory required at any given time.

C. Inside the GENPACK generations

The GENPACK strategy involves the dispatch of containers and their following migration into the *young* and the *old* generations, according to the informations gathered during the automatic profiling phase. Figure 11 shows the migrations occurring, during the first 2 hours, from/to the generations when DOCKER SWARM uses the GENPACK strategy. We complement these results by looking at the total

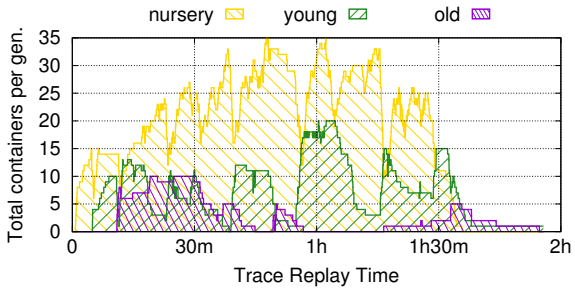

Figure 11. Migration of containers between generations.

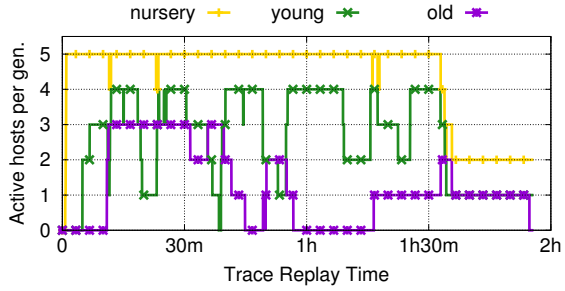

Figure 12. Active hosts per generations.

number of active hosts for each generation, as shown in Figure 12. The sampled Borg trace triggers 131 migrations from the nursery to the young and 50 from the young to the old generation. These results partially derive from the chosen configuration of monitoring periods. We postpone to future work a full sensibility analysis of these parameters with respect to the Borg trace.

While performing these experiments, we observe different replay timings—*i.e.*, the time required to completely inject the Borg trace in our cluster—between the scheduling strategies under test. Given the ideal duration of 1 hour, the *random* strategy completes in 1h19m54s, *spread* in 1h02m42s, *binpack* in 2h22m5s and finally the GENPACK strategy in 2h37m42s. These differences can be explained by the different load on the DOCKER daemon running on the host VMs and in general the ability to load balance the containers across the hosts and VMs. It is important to stress that these results correspond to the costs of injecting the Borg trace with our prototype, but do not directly reflect the system costs of scheduling in real conditions. In particular, as we show in the following Section VII-D, the four strategies are equivalent with respect to job completion times.

D. Job completion time

We compare the observed job completion time when using the default SWARM strategies against the GENPACK strategy. Figure 13 shows that our approach does not impact negatively the executing time of the jobs. The tested strategies result in the same long tail of few longer jobs as well as the same inflection point for the 90th percentile. Instead, the

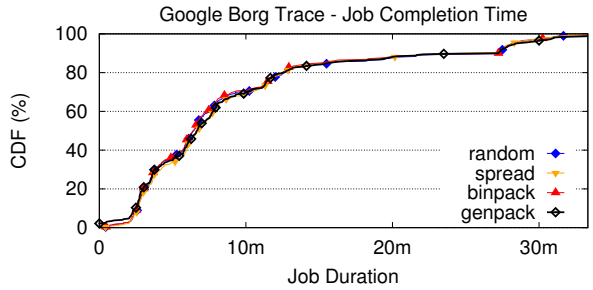

Figure 13. Distribution (CDF) of job completion times.

4 strategies produce the same job completion distribution, and thus offer the same experience to the end-users of a GENPACK cluster. Given the reported job completion times, we can conclude that GENPACK does not over-commit the cluster resources and rather offers a resource-efficient scheduling approach.

E. Energy impact

We demonstrate the interest of adopting the GENPACK strategy for a cloud data center by comparing its energy impact to the default SWARM strategies. We rely on BITWATTS probes to continuously report on the container's and node's power consumption. Figure 14 shows our results. We present the normalized results against the `spread` baseline. While the `binpack` strategy saves up 9% of energy compared to `spread` default built-in strategy, GENPACK outperforms the existing strategies by saving 23% of the cluster consumption. These impressive results are due to the capability of GENPACK of *i)* packing efficiently system containers onto a reduced number of nodes per generation and *ii)* turning off unused nodes in each of the generations. This result suggests that the GENPACK approach can lead to sensible savings for cloud data centers. In particular, our evaluation based on real-world traces considers a large diversity of jobs' durations and profiles as well as incoming workloads, even though we could not inject the full Google Borg Trace.

We can also observe that the deployment of additional containers for monitoring the resource consumptions and computing the container envelopes does not penalize the power usage efficiency of GENPACK. We can therefore conclude that GENPACK can achieve the same performances as existing scheduling strategies of DOCKER SWARM, but at a drastically reduced cost.

VIII. RELATED WORK

Resource management and scheduling is an important topic. Many researchers have addressed various aspects of scheduling resources during the last decades. Scheduling has been addressed in the context of GRID computing [19], distributed systems [20], HPC [21], batch processing [22], MapReduce [23], and more recently in the context of VM [24] and container scheduling [25] in large clusters.

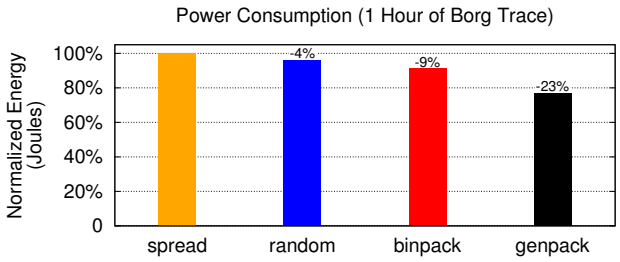

Figure 14. Normalized energy consumption.

Distributed job schedulers like the CONDOR scheduler [20] performs a *match making* between a job waiting to run and the machines available to run jobs. Hence, each job explicitly describes its resource requirements and also a *rank expression* that permits the scheduler to select the machine that is most suited to run this job. Also, the resources of a machine have to be explicitly described. In GENPACK, we avoid the need to describe jobs and machines by performing an automatic profiling of the containers and nodes (cf. Section IV).

The OPENSTACK NOVA scheduler does not consider CPU load for the assignment of VMs [24]. The scheduling in OPENSTACK, no matter the selected strategy, is rather based on statically defined RAM and CPU size of the VM, known as flavors [24]. In our experience, the simple round-robin scheduler results in many cases in situations where all hosts run some VMs and none of the hosts can be switched off to reduce the energy consumption (cf. Section II).

OPTSCHED [26] compares the energy implications of a *round robin* scheduler, a *first fit* scheduler, and an *optimized* scheduler that knows the run time of (some of) the VMs upon scheduling. Knowing the run times before starting a VM helps reduce the total energy consumed by a cluster. In GENPACK, however, run times are not known *a priori* and GENPACK is able to automatically learn the profile that is used by the scheduler along generations to improve the energy efficiency of the cluster (cf. Section VII).

YARN [23] is a two-level scheduler that can handle multiple workloads on the same cluster. It is request-based and supports locality of scheduling decisions such that jobs can, for example, access data on local disks to avoid remote accesses via the network. Nonetheless, the scheduling in YARN implements a strategy close to the *spread* strategy of DOCKER SWARM, thus suffering from the same limitations in terms of power usage efficiency.

Google developed a series of container management systems during the last 10 years [25]: BORG, OMEGA, and more recently KUBERNETES. Initially, Google started with a centralized container management system called BORG, which remains the main system in use by Google [7]. OMEGA is based on the lessons learned from BORG and has a principled architecture that includes a centralized transactional store and an optimistic concurrency control. In particular, the OMEGA architecture supports multiple concurrent schedulers. Finally, KUBERNETES is an open source container system that focuses on simplifying the task of application developers and has less focus on maximizing the utilization of clusters—which is the focus of OMEGA and BORG. Compared to GENPACK, all these approaches does not incorporate the concept of generations within the cluster to automatically learn about the container profiles at runtime.

DOCKER SWARM is very similar to KUBERNETES in that it aims to support *cloud native* applications. SWARM permits users to define applications consisting of a set of containers. The focus is on simplifying the typical tasks of the application developers like load balancing, elasticity, and high availability. Unlike GENPACK, the main goal of Swarm is not on ensuring a high utilization of a compute cluster, but this paper demonstrates how we succeed to extend it in order to address this concern.

IX. CONCLUSION

Efficient VM or container scheduling is particularly critical in cloud data centers to not only provide good performance, but also minimize the hardware resource required for running concurrent applications. This can, in turn, reduce the costs of operating a cloud infrastructure and, importantly, reduce the associated energy footprint. In particular, when efficiently packing containers on physical hosts, one can save significant amounts of energy by turning off unused servers.

In this paper, we propose GENPACK, a new scheduler for containers that borrows ideas from generational garbage collectors. An original feature of GENPACK is that it does *not* assume the properties of the containers and workloads to be known in advance. It relies instead on runtime monitoring to observe the resource usage of containers while in the "nursery". Containers are then run in a young generation of servers, which hold short-running jobs and experience relatively high turnaround. This collection of servers can also be elastically expanded or shrunk to quickly adapt to the demand. Long-running jobs are migrated to the old generation, which is composed of more stable and energy-efficient servers. The containers in the old generation run for a long time and typically experience relatively even load, hence they can be packed in a very efficient way on the servers without need for frequent migrations.

We have implemented GENPACK in the context of DOCKER SWARM and evaluated it using a real-world trace. Our comparison against SWARM's built-in schedulers shows that GENPACK does not add noticeable overheads while providing more efficient container packing, which can result in important energy savings.

Our perspectives for GENPACK includes a careful sensitivity analysis of key parameters like the k-means value or the scheduling period. We also plan to evaluate the performances of GENPACK in a long-running deployment

evolving not only CPU- and memory- intensive containers, but also network- and disk-intensive ones.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Commission, Information and Communication Technologies, H2020-ICT-2015 under grant agreement number 690111 (SecureCloud project). This work was partially supported by a grant from CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

REFERENCES

[1] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," *SIGPLAN Not.*, vol. 19, no. 5, pp. 157–167, Apr. 1984.

[2] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, no. 6, pp. 419–429, Jun. 1983.

[3] Google, "cAdvisor." [Online]. Available: https://github.com/google/cadvisor

[4] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, "Process-level power estimation in vm-based systems," in *EuroSys*. ACM, 2015, pp. 14:1–14:14.

[5] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, 2007.

[6] Docker, "Swarm." [Online]. Available: https://www.docker.com/products/docker-swarm

[7] A. Verma, L. Pedrosa, M. Korupolu, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*. ACM, 2015, pp. 18:1–18:17.

[8] Ubuntu, "Stress-ng." [Online]. Available: http://kernel.ubuntu.com/~cking/stress-ng

[9] influxdata, "InfluxDB." [Online]. Available: https://www.influxdata.com/time-series-platform/influxdb

[10] Swipely, "Docker API." [Online]. Available: https://github.com/swipely/docker-api

[11] Docker, "Docker Remote API." [Online]. Available: https://docs.docker.com/engine/reference/api/docker_remote_api

[12] Resque, "Resque." [Online]. Available: https://github.com/resque/resque

[13] libvirt, "host-passthrough." [Online]. Available: https://libvirt.org/formatdomain.html

[14] Docker, "Scheduler Strategies." [Online]. Available: https://docs.docker.com/swarm/scheduler/strategy

[15] OpenNebula, "Scheduler." [Online]. Available: http://docs.opennebula.org/5.0/operation/host_cluster_management/scheduler.html

[16] J. Wilkes, "More Google cluster data," Google research blog, Nov. 2011, posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[17] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2012.03.20. Posted at http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[18] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[19] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol. 1. IEEE, 2000, pp. 283–289.

[20] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: a distributed job scheduler," in *Beowulf cluster computing with Linux*. MIT press, 2001, pp. 307–350.

[21] D. Jackson, Q. Snell, and M. Clement, *Core Algorithms of the Maui Scheduler*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–102.

[22] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, vol. 2. IEEE, 2005, pp. 776–783.

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[24] O. Litvinski and A. Gherbi, "Openstack scheduler evaluation using design of experiment approach," in *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. IEEE, 2013, pp. 1–7.

[25] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.

[26] T. Knauth and C. Fetzer, "Energy-aware scheduling for infrastructure clouds," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE, 2012, pp. 58–65.