Department of Electrical and Computer Engineering
University of Thessaly,
Volos,Greece

Μηχανισμοί παρακολούθησης πληροφορίας επίδοσης και ελέγχου της διαμόρφωσης του συστήματος σε συστήματα εικονικών μηχανών / Performance monitoring and configuration control mechanisms on virtualized systems

Athanasios Gkantsidis

Supervisors:
Christos D. Antonopoulos, Assistant Professor
Spyros Lalis, Associate Professor

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to thank my advisors Christos D. Antonopoulos and Spyros Lalis for their help and guidance throughout this work. It is due to their inspiration and continuous encouragement that I was able to successfully complete this work. Also I would like to thank my family for their continuous support throughout these years and my colleague and friend Rania Tsilomitrou for all the moral and psychological support.

ABSTRACT

Recent developments in virtualization technologies, have led to the adoption of virtualization server, which increases the workload of datacenters. Increasing the workload on the servers in a data center, we also increase energy consumption. Prior work [1] has shown that, the cost of powering the servers on a datacenter, is 30% of the total cost of a datacenter. Associated costs have shifted the research focus from optimizing performance to finding a tradeoff between performance and energy efficiency.

Quantifying the power consumption of individual applications running in parallel on a server, is a key component to administrators of datacenters to extract useful conclusions and be able to implement policies to reduce consumption at peak times. Furthermore, this will help the companies to design their billing policy based on power consumption of each virtual machine.

The main purpose of this thesis was to describe the API extensions implementation process on Libvirt. We have implemented a Libvirt extension that extracts a number of useful counters, which are useful for an application that uses these counters in order to estimate the power consumption of every running virtual machine. Also we have implemented Libvirt API extensions enabling host configuration. More specifically, we offer Libvirt users the functionality to control frequency in order to reduce energy consumption.

ΠΕΡΙΛΗΨΗ

Οι πρόσφατες εξελίξεις σε τεχνολογίες virtualization εχουν οδηγήσει στην υιοθέτηση virtualization server οι οποιοι αυξάνουν το φόρτο εργασίας ενος datacenter. Αυξάνοντας το φόρτο εργασίας στους servers σε ένα κέντρο δεδομένων (datacenter) παράλληλα αυξάνεται και η κατανάλωση ενέργειας. Έρευνες [1] δείχνουν ότι το κόστος τροφοδοσίας ενός datacenter είναι περίπου το 30% του συνολικού κόστους του datacenter. Το κόστος έχει μετατοπίσει την εστίαση της έρευνας από τη βελτιστοποίηση της απόδοσης στην αναζήτηση καποιας ισορροπίας μεταξύ της απόδοσης και της ενεργειακής απόδοσης.

Η ποσοτικοποίηση της κατανάλωσης ενέργειας των επιμέρους εφαρμογών που τρέχουν παραλληλα σε ενα server ειναι βασική παραμετρος για να μπορουν οι διαχειριστές ενος datacenter να εξάγουν χρήσιμα συμπεράσματα και να μπορούν να εφαρμόσουν πολιτικές για μείωση της κατανάλωσης σε ώρες αιχμής ή ακόμα και για να εξάγουν συμπεράσματα για τη χρέωση των υπηρεσιών τους.

Βασικός σκοπός της διπλωματικής ήταν η περιγραφή της διαδικασίας προσθήκης λειτουργικότητάς στη βιβλιοθήκη Libvirt. Οι προσθήκες αφορούν μια επέκταση της βιβλιοθήκης Libvirt για την εξαγωγή μετρικών που χρησιμοποιήθηκαν σε μια εφαρμογή για τον υπολογισμό της κατανάλωσης ενέργειας εικονικών μηχανών που λειτουργούν παράλληλα στο ίδιο εξυπηρετητή,καθώς και επεκτάσεις του API της Libvirt για τη διαμόρφωση του host ώστε να ελέγχουμε τη συχνότητα λειτουργίας του επεξεργαστή με στόχο την μείωση κατανάλωσης ενέργειας.

CONTENTS

LIST OF FIGURES

LISTINGS

1

INTRODUCTION

1.1 PROBLEM DESCRIPTION

The continuing demand for power within the datacenter industry, has received significant amounts of press in recent years. These energy intensive industries, are signing contracts with electricity suppliers, in order to be protected from unexpected energy price increases and in order to guarantee a low fixed rate for the full term of the energy supply contract. Power consumption is a continuously changing number, which in peak hours, may be bigger than the energy that the datacenter itself provides or exceeds the agreed contracted level. As a consequence, penalties from the energy suppliers may be applied. To avoid this phenomenon,datacenter administrators should use techniques such as job suspension and cpu frequency scaling.For a successful implementation of these techniques, it is important to use a system that calculates the consumption of each virtual machine, in order to be able to disable or to migrate virtual machines to another server, to ensure the proper function of the system.

The problem that the scientific community is called to solve, is the attribution of the socket power consumption to the applications that run simultaneously on server. Because of the fact that the Linux hypervisor perceives as regular process the virtual machines [5], we could handle the problem of calculating the consumption of virtual machines as we do in the case of simple applications. There are two approaches to solve the above problem. The first approach refers to cpu counters that approximate power consumption with statistical models and the second one refers to measurement of the power supply and proper attribution of the power consumption to the applications, through the cpu usage percentage [6].

In this thesis, we develop a system that calculates the energy consumption per virtual machine by using Libvirt library, so that the administrators do not need to use specialized tools to monitor the consumption. Additionally, we could use the information of the consumption of a VM programmatically for implementing policies or other monitoring applications.

The main purpose is to describe the API extensions implementation process on Libvirt. The extensions concerns the ex-

traction of the individual consumption of a co-running virtual machine and the implementation of knobs, in order to configure host system.

For the calculation of the socket power consumption, we use statistical models[4]. Moreover, we calculate the power consumption per virtual machine using the model that finds the cpu-cycles of every virtual machine. For the host configuration system , we change sysfs values to configure the driver cpufreq or the intel p-state driver, in order to be able to change the frequency or to set a power limit over a time window.

1.2 THESIS STRUCTURE

This thesis consists of three parts. The first part deals with background issues. More specifically, in section 2.1 we analyze the architecture of Qemu/KVM,in section 2.2 we discribe Libvirt architecture, in section 2.3 we describe Intel RAPL in order to extracts the counters of socket power consumption and finally in section 2.4 we describe the Perf profiler.

Second part describes in detail, the development of Libvirt API extensions that we added. This provides methodology to developers, who want to add new features to Libvirt API. Also, we implement an application that uses our extension and extracts the power consumption per virtual machine. Next,in Chapter 4, we evaluate our model by using benchmarks. Finally,in Chapter 6, we describe the conclusions and discuss possible future work.

2

BACKGROUND

For better understanding of the overall procedure of Libvirt API extension development, in this chapter we describe Qemu/KVM hypervisor, Libvirt API and Perf profiler. Finally, we analyze the main feature of Intel's RAPL.

2.1 QEMU/KVM

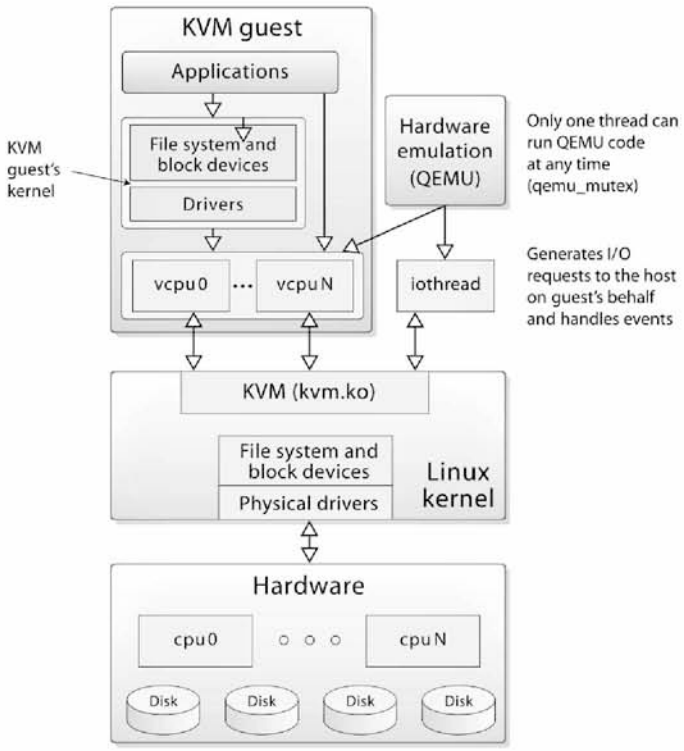

Figure 1: Architecture of the Kernel-based Virtual Machine (KVM) [2]

KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware, containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko.

KVM focuses on functionalities of lower levels of a virtual machine. Such functionalities could be processor registers, Mem-

ory Management Unit[4] and controlling switches from guest to host in hardware.

Qemu[7] is a processor emulator that is able to emulate a number of processor architectures and deals with handling the emulated hardware such as hard disks, network cards, audio interfaces and USB devices. Each core uses a separate instance of Qemu binary translation engine, with a thin library layer to handle the inter-core , the device communication and synchronization. Qemu communicates with the KVM module through the /dev/kvm interface through a series of ioctls. Listing 2.1 shows the basic flow of the KVM module communication inside Qemu. First opening the interface and issuing the correct ioctl to create a new guest. Then the guest accesses a hardware device register, halts the guest CPU, or performs other special operations, KVM exits back to Qemu, Qemu handles the exit and emulates the desired outcome. Each guest virtual CPU is a Qemu thread that is being scheduled by the OS scheduler as a regular process. A dedicated iothread runs a select(2) event loop to process I/O such as network packets and disk I/O completion. The memory of a guest is allocated by Qemu at launch, and is mapped into the address space of the Qemu process. This acts as the physical memory of the guest.

Listing 2.1: Basic flow of a guest CPU

```
open ("/dev/kvm")
ioctl (KVM CREATE VM)
ioctl (KVM CREATE VCPU)
for( ; ; ) {
    ioctl (KVM RUN)
    switch ( exit reason ) {
        case KVM EXIT IO : . . .
        case KVM EXIT HLT: . . .
    }
}
```

2.2 LIBVIRT LIBRARY

According to the website [8] Libvirt is a collection of software that provides a convenient way to manage virtual machines and other virtualization functionality, such as storage and network interface management. These software pieces include an API library, a daemon (libvirtd), and a command line utility (virsh). A primary goal of libvirt is to provide a single way to manage multiple different virtualization providers/hypervisors.

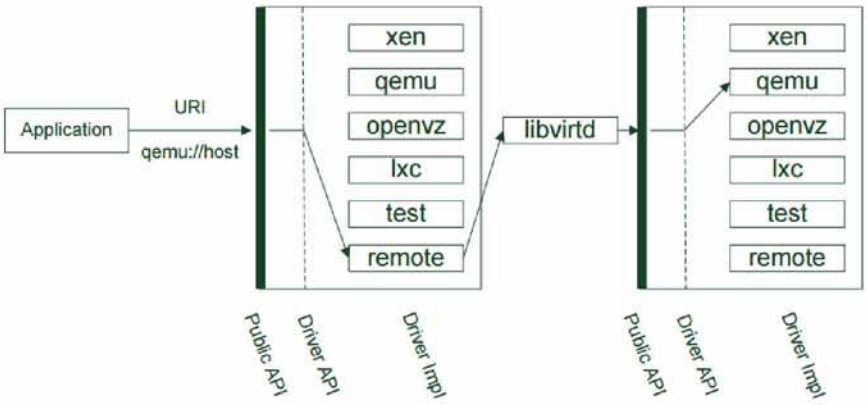

Figure 2: Libvirt Architecture

Libvirt, as we can see in Figure 2, consists of two parts, a public API for applications that use the library and a driver API, that contains the drivers, which are useful in order to communicate with various hypervisors. The drivers implement an API to communicate with Libvirt that matches the API they use to communicate with hypervisors and the API that hypervisors use to communicate with Libvirt. The goal is to call a function that starts a virtual machine with the same way, regardless of the hypervisor and the driver that we use. However, when an external application communicates with the Libvirt, it uses an URI, that defines ,via the virInitialize API, which driver to use.

Furthermore, Libvirt offers remote managment facilities by implementing a remote driver on the client and a deamon for handling requests from the server side, called libvirtd. Requests from a client are tunneled through the remote driver to the server, where the specific hypervisor is running. The libvirtd on the server receives the requested commands and locally calls the specific driver. The last piece of Libvirt software is virsh, which is a virtualizations shell built on top of Libvirt. This shell permits use the libvirt functionality, but in an interactive (shell-based) fashion. Finally, Libvirt uses xml files to configure the virtual machines. This configuration could be permanent, which indicates that it will remain the same after a

restart of the domain, or it could be temporary, which means that maintains the configuration only during the session.

2.3 RUNNING AVERAGE POWER LIMIT

Intel RAPL[4] provides counters that show the socket power consumption. Nevertheless, it is not an analog power meter and for this reason the estimation of the energy is being calculated approximately, by using performance counters and I/O models.

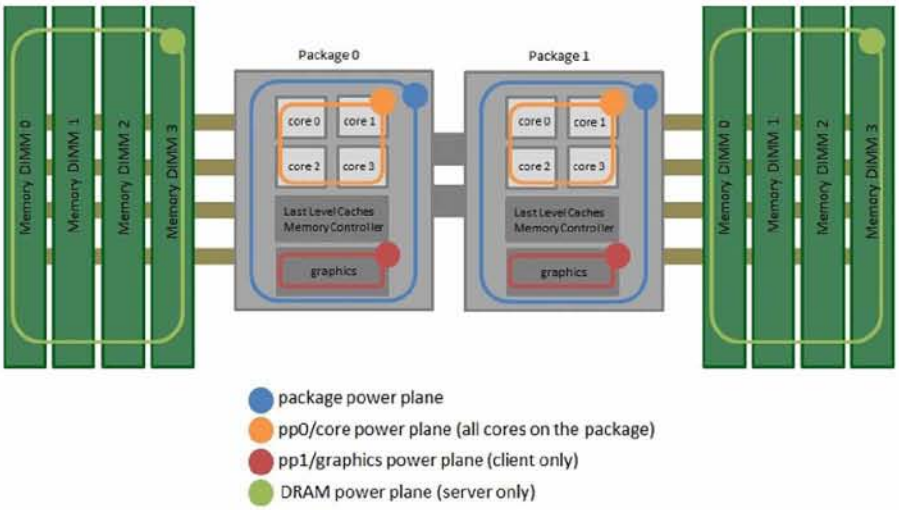

Figure 3: RAPL power domains [3]

As shown in Figure 3, the counters are divided into domains for precise control. These domains include core power plane, which refers to the energy that the cores are consuming and graphics power plane, which refers to the energy that embedded graphic card is consuming. Both these domains are parts of the package power plane that refers to the socket. Finally, the DRAM power plane refers to the DRAM energy consumption. We can access these hardware counters through /sys/fs [9]. Another RAPL's functionality, is the ability to limit the average consumption of the socket at a time window.

As we can see from Figure 4, the fields of register MSR_PKG_POWER_LIMIT, include the pkg Power Limit , which refers to the limit of the average energy consumption, at the time period we defined at the field time_window_power_limit. There are two zones that we can customise by putting different limits in each one.

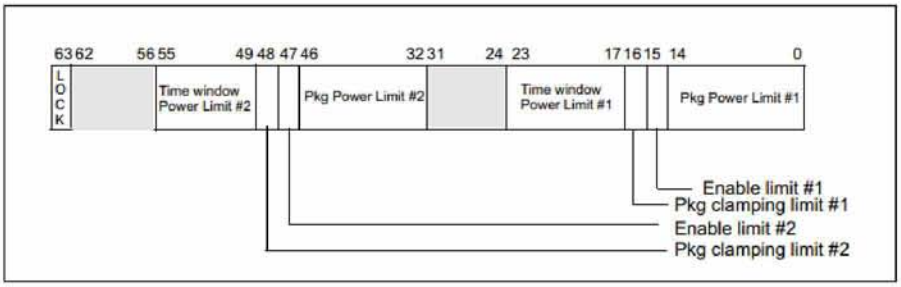

Figure 4: MSR_PKG_POWER_LIMIT Register [4]

2.4 LINUX PERFORMANCE EVALUATION TOOL

With the advent of technologies such as Symmetric Multi Processing (SMP) and Non-Uniform Memory Access (NUMA) developers notice issues with the performance of their programs because of the new hardware. Thus, Intel developed the PMU[10], a special unit providing information through counters in microarchitecture level like the number of machine cycles or even the number of instructions executed on the CPU. In Linux there are several interfaces that provide access to these counters. The perf_event [11],a key tool of the kernel, is the interface that we are going to use during the development. There are two modes that perf event counts events. The first calculates the collective events during the period of profiling program operation and the second mode creates a hardware interrupt after a specific number of events. In our case we use the first mode. The following function enables the perf event described by the first argument which is a struct perf_event_attr.

Listing 2.2: perf_event_open syscall

```
int perf_event_open(struct perf_event_attr *attr, pid_t
    pid, int cpu, int group_fd, unsigned long flags);
```

The second argument is the profiling application thread id, the third argument limits counting events in a particular kernel thread. Finally the group_fd is the file descriptor that we use if we want to count events from a group of threads. Since there is not a libc wrapper for the particular syscall we use the function syscall() with first argument the NR_perf_event_open.

2.5 POWER ESTIMATION MODEL

Model for attributing power consumption of each running virtual machine is as follows:

$$Power(Task_i) = TotalActivePower * \frac{Cycle(Task_i)}{\sum_{j=1}^{m} Cycle(Task_j)} \quad (1)$$

Model (1)[1] is not hyperthreading aware and is based in hypothesis that the power consumption of a task is proportional to the amount of computation CPUs perform for that task. We can estimate the amount of CPU computation using hardware events, such as CPU cycles and instructions. Authors examined other metrics, including instruction count, last-level cache reference and miss, through a wide range of microbenchmarks and conclude that non-halted cycle is the best to correlate power. Therefore, the virtual cpu threads will be pinned to separate physical cores, which will be selected during the creation of the xml configuration file of each virtual machine. Consequently, we need to calculate the total power consumption of the socket (Total_Active_Power) and also the cpu-cycles, that each virtual machine has consumed. These are estimated by using the two events that we added in Libvirt.

3

IMPLEMENTATION

In this chapter, we will describe the process of implementing a new API in Libvirt. We will add further functionality to perf-events API , which is implemented by Intel,that triggers perf-events, in order to monitor cpu-cycles per VM for virtual machine's qemu threads and socket 's power consumption. Also, we will implement API extensions for host configuration about cpu frequency scaling.The above changes will be made in version 2.0.0 of Libvirt.

3.1 DEFINE PUBLIC API

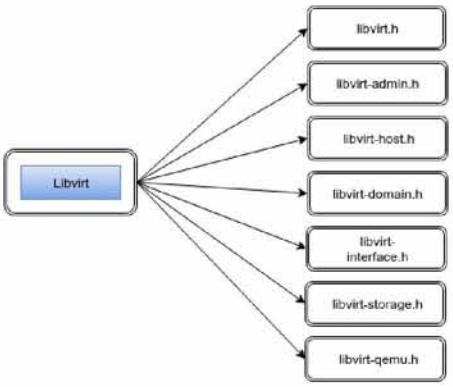

Figure 5: Libvirt domains

Firstly, we should define the public API. In case our changes contains new functions, we should import these functions into the Libvirt library headers and also import the export symbols, in order to make the new function visible to external programs. Developers as shown in Figure 5, have divided the library in order to be able to search code more conveniently. In our case, since the extension refers to the host, we add functions to /include/libvirt/ libvirt-host.h .

We implement the functions of public API in file libvirt-domain.c. The implementation of the public API according to the site[12] is largely a formality in which we wire up public API to the internal driver API. The public API implementation takes care of some basic validity checks before passing

control to the driver implementation. In RFC 2119 vocabulary, this function:

1. Should log a message with VIR_DEBUG() indicating that it is being called and its parameters.
2. Must call virResetLastError()
3. Should confirm that the connection is valid with virCheck-ConnectReturn() or virCheckConnectGoto()
4. If the API requires a connection with write privileges, MUST confirm that the connection flags do not indicate that the connection is read-only with virCheckReadOnly-Goto()
5. Should do basic validation of the parameters that are being passed in, using helpers like virCheckNonNullArgGoto()
6. Must confirm that the driver for this connection exists and that it implements this function
7. Must call the internal API
8. Should log a message with VIR_DEBUG() indicating that it is returning, its return value, and status.
9. Must return status to the caller.

Listing 3.1: virNodeSetGovernor function

```
int
virNodeSetGovernor(virConnectPtr conn,
                   int core,
                   int governor)
{
    VIR_DEBUG("conn=%p, core=%d, governor=%d", conn, core,
        governor);

    virResetLastError();

    virCheckConnectReturn(conn, -1);
    virCheckNonZeroArgGoto(governor, error);

    if (conn->driver->nodeSetGovernor) {
        int ret;
        ret = conn->driver->nodeSetGovernor(conn, core,
            governor);
        if (ret < 0)
            goto error;
        return ret;
    }
```

```
    virReportUnsupportedError();

 error:
    virDispatchError(conn);
    return -1;
}
```

3.2 IMPLEMENT THE INTERNAL DRIVER METHODS

The core of the process is to add the new functionality in the drivers of Libvirt. The purpose of Libvirt, as we mentioned in section 3.1, is to make the functionalities general for all the hypervisors and to be able to use them in the same way regardless of the different underlying implementation. We add our function to header file of hypervisor /src/driver-hypervisor.h and also, we add the respective fields to struct virHypervisorDriver , in order to declare that our functions refer to hypervisor functionality. The next step is to implement the functions at a specific hypervisor drivers. In our case, since we will add functionality to qemu, we have to add our functions to the file /src/qemu/qemu-driver.c and the appropriate fields to struct qemuHypervisorDriver as well.

Furthermore, if we add new files with internal functions which driver functions use ,we add the names of the files to src/makefile.am, for the purpose of being included in the compile and lastly we add symbols in the /src/libvirt_private.syms. The files that contain the main functionality of our API extensions are the following: src/util/virperf.h, src/util/virperf.c ,/src/virhostcpu.h,/src/virhostcpu.c .

For the implementation of host configuration extentions, we add three functions to the file virhostcpu.c. The first one, virHostSetFrequency, aims to change the operating cpu frequency of the corresponding physical core and has two arguments. The first argument is the id of a physical core and the second one is a string that contains the value of frequency. We use the interface of sysfs to change the value of scaling_setspeed. This specific functionality demands the driver of cpu-freq to be activated and governor "userspace" to be enabled. The next function, virHostSetGovernor, aims to change the governor, either cpu-freq or intel- pstate driver is being used. As far as cpufreq is concerned, we have two choices of governors, "ondemand" and "userspace". As for the intel-pstate, there are also two choices of governors, "powersave" and "performance". The last function that we implemented, is virHostSetRaplLimit, which has two ways to reduce the host's energy consumption. The first way uses the intel-pstate driver and changes the value of

max_perf_pct through sysfs. Intel - pstate driver does not gives us the ability to choose a specific operating cpu frequency. According to the workload, it chooses, each time, a specific p-state. Max_perf_pct limits the maximum P-State that will be requested by the driver. It states it as a percentage of the available performance. The second way uses the power caping framework. As we have mentioned in the section 2.3, we set a limit of average consumption for a specific time window.

In order to add functionality to the existing API of perf-events, we expand the function virPerfRdtEnable. In order to add the first event, providing that there is no wrapper at libc for the specific syscall, we use the function syscall, with parameters NR perf event open, to call the function we need, a structure rdt_attr as detailed below, an ID thread that we want to monitor and finally, a kernel thread ID, which in our case is -1, and indicates that it monitors all the kernel threads that the specific thread runs. The function syscall, returns a file descriptor through which we will be able to read the counter value. The struct rdt_addrt provides configuration for the event being created. The struct fields that we change are, the general type of the event,that could be software or hardware, the hardware event ID that we want to enable, which in our case is the cpu cycles, and lastly the inherit field, which if is enabled, the counter includes also the events of the children of the main thread. Considering that a vm consists of more than one threads, the inherit field should be enabled. The disabled field refers to whether the event will begin once we activate it or later. In our case, we prefer to initiate it manually. In order to activate the event we created, we call the function ioctl and change the enable state of the event in the Libvirt, as well.

Listing 3.2: virPerfRdtEnable function

```
if (event->type == VIR_PERF_EVENT_CYCLES){
        memset(&rdt_attr, 0, sizeof(rdt_attr));
        rdt_attr.size = sizeof(rdt_attr);
        rdt_attr.type = PERF_TYPE_HARDWARE;
        rdt_attr.config = PERF_COUNT_HW_CPU_CYCLES;
        rdt_attr.inherit = 1;
        rdt_attr.disabled = 1;
        rdt_attr.enable_on_exec = 0;

        event->fd = syscall(__NR_perf_event_open,
            &rdt_attr, pid, -1, -1, 0);
        if (event->fd < 0) {
            virReportSystemError(errno,
                       _("Unable to open perf type=%d for
                           pid=%d"),
                       event_type, pid);
            goto error;
        }

        if (ioctl(event->fd, PERF_EVENT_IOC_ENABLE) < 0) {
            virReportSystemError(errno,
                       _("Unable to enable perf event for
                           %s"),
                       virPerfEventTypeToString(event->type));
            goto error;
        }

        event->enabled = true;
        return 0;

    }
```

As for the second event that we want to activate, there are some differences. We begin to read the type of the event from the sysfs and then we create the event with the call of the function syscall. The only difference in comparison with the first event is that there is neither a specific thread ID nor a physical core, as the counter is not only for a specific kernel thread but generally for the socket.

3.3 IMPLEMENT THE RPC CLIENT

The next step is to implement the remote protocol. If our functions return a different value of 0 or 1, then we need to create two structs in /src/remote_protocol.The first struct will be used to describe the arguments that need to be passed to the remote function and the second one will be used for the returned values from this function.

Listing 3.3: remote_protocol_structs function

```
struct remote_domain_set_frequency_args {
        int                     core;
        remote_string           frequency;

};
struct remote_domain_set_rapl_limit_args {
        int                     value;
        remote_string           type;
        remote_string           type_value;
};

struct remote_domain_set_governor_args {
        int                     core;
        int                     governor;
};


enum remote_procedure {
        .
        .
        .
        REMOTE_PROC_NODE_SET_FREQUENCY = 374,
        REMOTE_PROC_NODE_SET_RAPL_LIMIT = 375,
        REMOTE_PROC_NODE_SET_GOVERNOR = 376,
};
```

Subsequently, we add values to the remote_procedure enum for each new function added to the API. Once these changes are in place, it's necessary to run 'make'(rpcgen) in the src directory to create the .c and .h files required by the remote protocol code. The remote method calls go in: src/remote/remote_driver.c . Each remote method invocation does the following:

1. Locks the remote driver;
2. Sets up the method arguments;
3. Invokes the remote function;
4. Checks the return value, if necessary;
5. Extracts any returned data;
6. Frees any returned data;
7. Unlocks the remote driver.

Listing 3.4: remoteNodeSetGovernor function

```
static int
remoteNodeSetGovernor(virConnectPtr conn, int core, int
    governor)
{
    int rv = -1;
    struct private_data *priv = conn->privateData;
    remote_node_set_governor_args args;

    remoteDriverLock(priv);

    args.core = core;
    args.governor = governor;

    if (call(conn, priv, 0, REMOTE_PROC_NODE_SET_GOVERNOR,
             (xdrproc_t)xdr_remote_node_set_governor_args,
                (char *)&args,
             (xdrproc_t)xdr_void, (char *)NULL) == -1) {
        goto done;
    }

    rv = 0;

done:
    remoteDriverUnlock(priv);
    return rv;
}
```

3.4 IMPLEMENT THE SERVER SIDE DISPATCHER

The next step is the implementation of server side dispatcher, that is simply a matter of deserializing the parameters passed in from the remote caller and passing them to the corresponding internal API function. The changes are implemented in: /daemon/remote.c After all three pieces of the remote protocol are complete, and the generated files have been updated, it will be necessary to update the file: src/remote_protocol-structs

Listing 3.5: remoteDispatchNodeSetFrequency function

```
static int remoteDispatchNodeSetFrequency(
    virNetServerPtr server ATTRIBUTE_UNUSED,
    virNetServerClientPtr client,
    virNetMessagePtr msg ATTRIBUTE_UNUSED,
    virNetMessageErrorPtr rerr,
    remote_node_set_frequency_args *args)
{
    int rv = -1;
    char *frequency;
```

```
    struct daemonClientPrivate *priv =
        virNetServerClientGetPrivateData(client);

    if (!priv->conn) {
        virReportError(VIR_ERR_INTERNAL_ERROR, "%s",
            _("connection not open"));
        goto cleanup;
    }

    frequency = args->frequency ? *args->frequency : NULL;

    if (virNodeSetFrequency(priv->conn, args->core,
        frequency) < 0)
        goto cleanup;

    rv = 0;

cleanup:
    if (rv < 0)
        virNetMessageSaveError(rerr);
    return rv;
}
```

3.5 VIRSH EXTENSION

Virsh, as we mentioned in section 2.2, is a virtualizations shell built on top of Libvirt. In order to complete the procedure, we will add a new command for every new function. The code of virsh is divided, according to which domain of Libvirt it refers to. Considering that the additions we made refer to host, we will add the new commands to the file /tools/virsh-host.c.

Listing 3.6: virsh frequency command function

```
/*
 * "frequency" command
 */
static const vshCmdInfo info_frequency[] = {
    {.name = "help",
     .data = N_("set frequency")
    },
    {.name = "desc",
     .data = N_("Set frequency of the specific core.")
    },
    {.name = NULL}
};

static const vshCmdOptDef opts_frequency[] = {
    {.name = "core",
     .type = VSH_OT_INT,
     .help = N_("core id")
    },
    {.name = "frequency",
     .type = VSH_OT_STRING,
     .help = N_("frequency number")
    },
    {.name = NULL}
};

static bool
cmdNodeSetFrequency(vshControl *ctl, const vshCmd *cmd)
{
    int core;
    const char *frequency = NULL;
    int result;

    virshControlPtr priv = ctl->privData;

    if (vshCommandOptInt(ctl, cmd, "core", &core) < 0)
        return false;

    if (vshCommandOptStringReq(ctl, cmd, "frequency",
        &frequency) < 0)
        return false;
```

```
    if ((result =
        virNodeSetFrequency(priv->conn,core,frequency)) < 0)
        return false;

    vshPrint(ctl, "%d\n", result);

    return true;
}
```

As it seems from Listing 3.6, we add a variable vshCmdInfo that contains informations on the command and a variable vshCmdOptDef for the parameters which the function needs,when it will be called. Finally, we add a command that contains the main functionality, and an entry on the table vshCmdDef, in order to be recognised by the shell.

For the additions we made to the existing API of perf-events ,we use the API of stats, which already exists, in order to return the counters we created in the previous steps. We add a field in enumeration virDomainStatsTypes in header file /include/libvirt/ libvirt-domain.h and also the IDs for the created events. We change the existing command domstats as well, so that it includes the new counters, in the results that returns.

3.6 POWER ESTIMATION APPLICATION

For the calculation of the consumption per virtual machine we have implemented the get_the_power function. The function arguments are a pointer to the connection with the corresponding Hypervisor that controls the virtual machines and a string with the name of the virtual machine that we are interested in. The function returns the corresponding energy consumption once the events are activated up to the time of the call. We use the function VirConnectGetAllDomainStats in order to return the number of cycles that every running virtual machine consumes and the socket total energy consumption. Moreover, this function calculates the sum of cycles that every virtual machine consumes and expresses the counter of the socket total energy consumption in joules by multiplying it by the scaling factor exposed in .scale file in /sys/ fs. Finally, we use 1 to find the power consumption for the virtual machine that we are interested in. As an example of usage of the function get_the_power, we implemented an application that calls the function periodically every n seconds and prints energy consumption in watt for all running virtual machines.

Listing 3.7: get_the_power function

```
double get_the_power(virConnectPtr conn, char
    *domain_name){


   virDomainStatsRecordPtr *test;

   int ret;
   unsigned int stats = VIR_DOMAIN_STATS_PERF;
   int flags2 = 0;
   int i,j;
   FILE *fff;
   double scale;
   char filename[BUFSIZ];
   time_t rawtime;
   double cycles[30];
   double power[30];
   double power_cnt;
   double sum_cycles;


   ret =
       virConnectGetAllDomainStats(conn,stats,&test,flags2);
   if (ret < 0)
      printf("Something going wrong...");

   sprintf(filename,"
   /sys/bus/event_source/devices/power/events/energy-cores.scale");
   fff=fopen(filename,"r");

   if (fff!=NULL) {
      if(fscanf(fff,"\%lf",&scale) > 0){
          fclose(fff);
       }
   }

   sum_cycles = 0;

   for(i = 0; i < ret; i++){


      for (j = 0; j < test[i]->nparams; j++) {
         if(strcmp(test[i]->params[j].field,"perf.cycles")
             == 0){
            cycles[i] = test[i]->params[j].value.ul;
            sum_cycles += cycles[i];
         }
         else{

            power_cnt = test[i]->params[j].value.ul;
```

```
            }
        }
    }

    for(i = 0; i < ret; i++){
        power[i] = (power_cnt * scale *
            (cycles[i]/sum_cycles));
            if(strcmp(virDomainGetName(test[i]->dom),domain_name)
                == 0){
                virDomainStatsRecordListFree(test);
                return power[i];
            }
            else{
                printf("Something going wrong...\n");
                virDomainStatsRecordListFree(test);
                return 0.0;
            }
    }
    return 0;
}
```

4

EXPERIMENTAL VALIDATION

4.1 EXPERIMENTAL SETUP

In this chapter we validate our model in virtualized systems, using benchmarks to stress the system and check the functionality in situations with heavy workload like datacenter environment. The experimental validation was carried out on a system equipped with one 4-core Intel i5 4460 processor, clocked at 3.2 GHz, with 8 GB DRAM and SSD with sequential read 540 MB/s and sequential write 490 MB/s. The operating system is Ubuntu 14.04, using the 4.2 Linux kernel. We will use virtual machines with 1/2/4 vcpus and 1 GB RAM with the same operating system.

We use the turbostat in order to get reliable measurements of socket consumption, as it is also uses the rapl counters. We use Lulesh [13] as cpu benchmark, so that we will be able to approach the cpu usage on production,the stream benchmark [14] as memory benchmark and also the iozone [15] as io benchmark. We poll the counters periodically every one second, so that joules be converted automatically to watt because of

$$P(W) = \frac{E(J)}{t(s)} \tag{2}$$

Also we use disjoint sets of physical cores for the virtual machines running in parallel.Every virtual cpu is pinned to a physical core, in order to calculate correctly the unhalted cpu cycles, that a virtual machine consumes.

4.2 CPU

Figure 6 shows the power consumption of the cores, when Lulesh runs on the host using four physical cores (4p3200), as well as the power consumption, when Lulesh runs in guest with four virtual cpus (v4p3200). We observe that the guest VM has nearly the same power consumption as the host, since both of them stretch the usage of each used core to its limit, but with worse performance. The performance degradation in the guest VM comes from the virtualization, but does not approaching

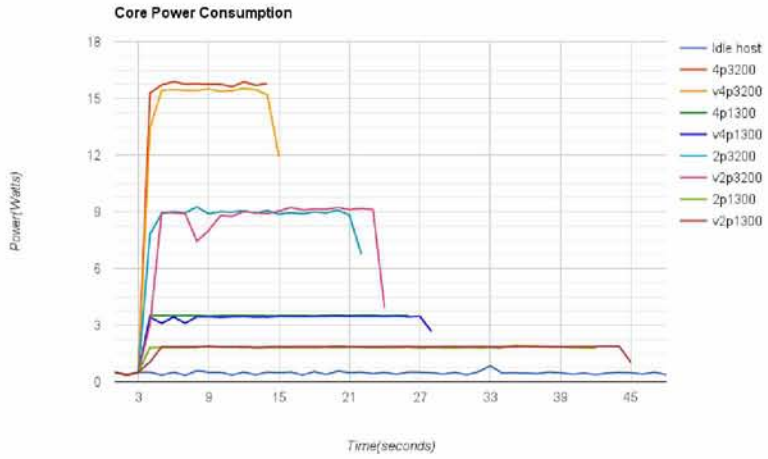

Figure 6: Cores power consumption(CPU)

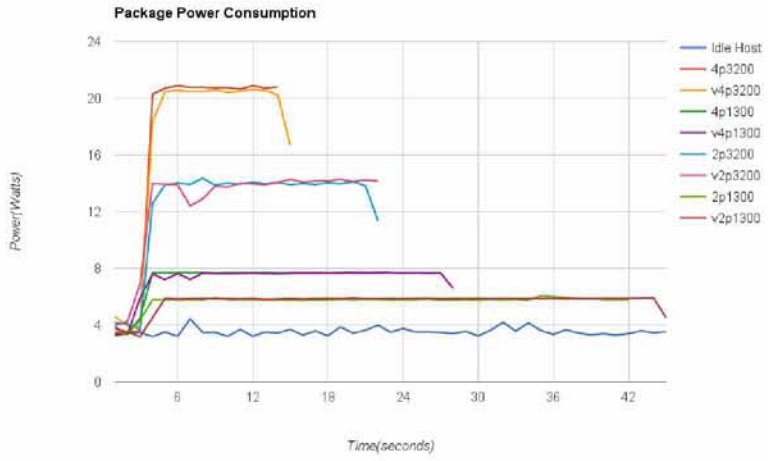

Figure 7: Package power consumption(CPU)

the overhead of a type 2 hypervisor. We have noticed same behavior in case of two cores (2p3200, v2p3200) and in case of reduced cpu frequency (1300mhz). Also, we noticed same behavior testing the system with floating point numbers workload. The processing of calculations is slow because the KVM checks whether the command is an interrupt, a page fault or a simple command, in order to decide whether to remain in guest mode or not [16, 17].

Figure 7 shows that the energy consumption of the package is identical to the power consumption of the cores because of minimal power consumption on dram and gpu power planes.

In Figure 8 we depict the power attribution, which derives from the application that we developed. We observe the general

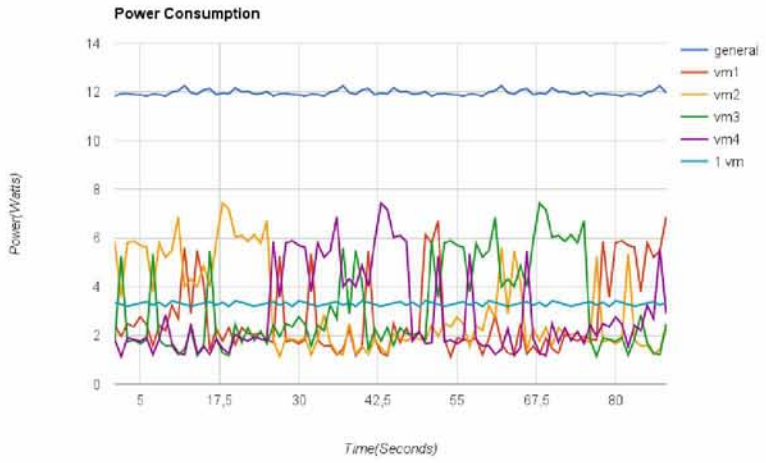

Figure 8: Power attribution (CPU)

consumption exported from rapl counters (general) ,the consumption of a VM with a vcpu running Lulesh(1vm) and the remaining four lines which indicate the power consumption attribution of four VM running alongside running Lulesh.We approximate correctly the consumption of a vm, depending on cpu usage(cpu-cycles).

4.3 MEMORY

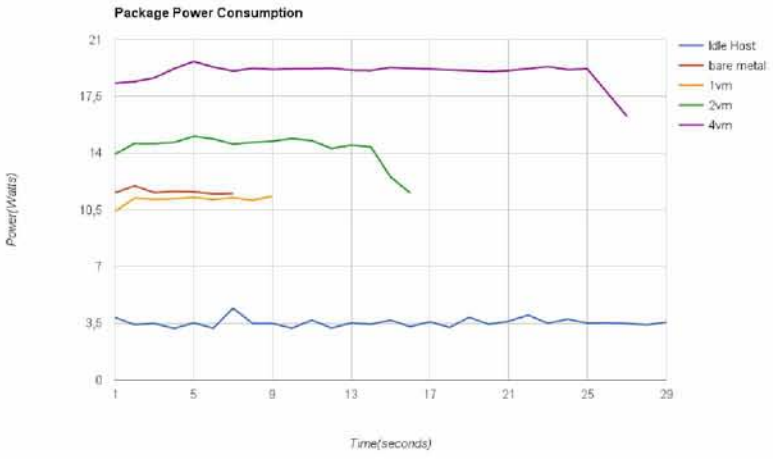

Figure 9: Package power consumption (RAM)

In Figure 9, we depict package power consumption in case that the stream benchmark, which is not multithreaded, runs on the host(bare metal) or runs in a single VM or in two/four

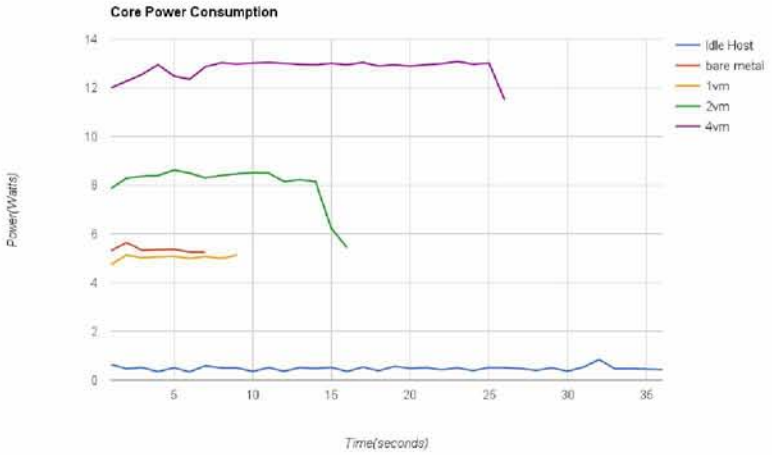

Figure 10: Cores power consumption (RAM)

VM running in parallel. An interesting note is that less power is consumed by the VM compared to the host for the same test but energy consumption is bigger because of performance degradation.Also the total power consumption remains nearly constant,regardless of memory usage. Furthermore, in case that the stream benchmark runs in four VMs in parallel, we notice an increase of power consumption, but also a decrease of performance, because of bus saturation.

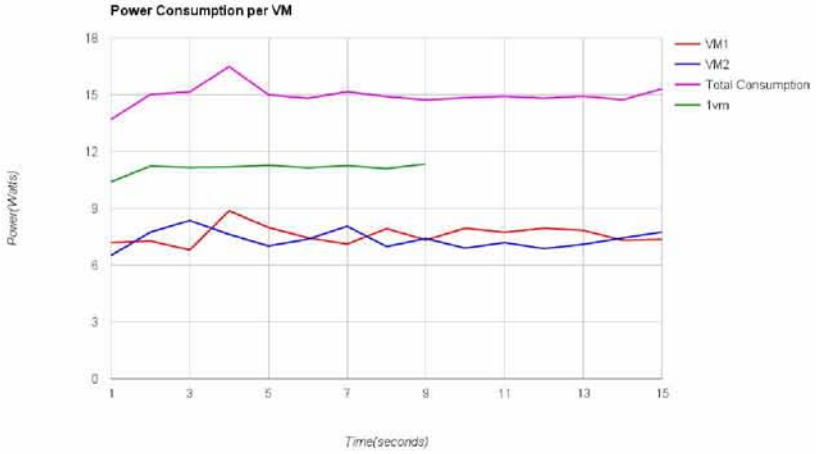

Figure 11: Power attribution (RAM)

In Figure 11, we depict power consumption per VM when the stream benchmark runs in two VM running in parallel. We are able to observe the same behavior as in the experiment of the previous graph, which is reduction in power consumption, reduction in yield but increase of energy consumption.

4.4 IO

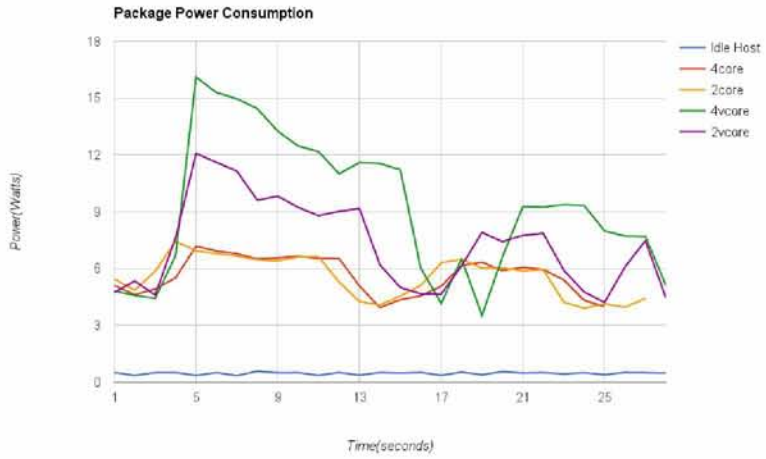

Figure 12: Package power attribution (IO)

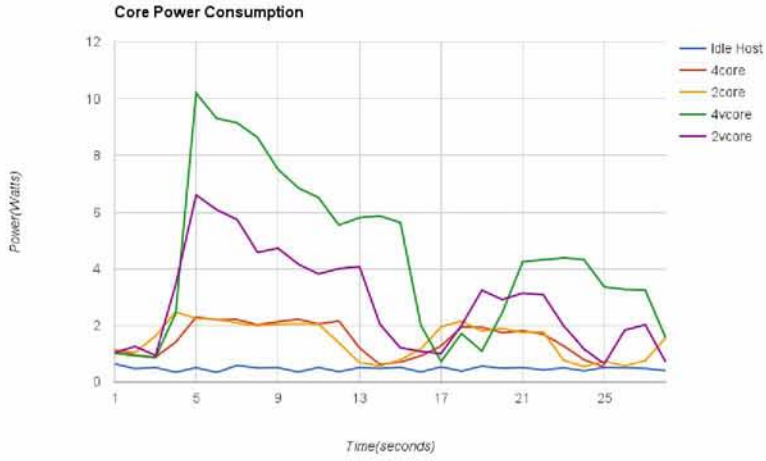

Figure 13: Cores power consumption (IO)

In Figure 12, we depict the power consumption in case that the iozone benchmark runs on the host, using two/four physical cores and in case that runs in guest VM with two/four virtual cpus.In case of guest VM, we noticed an increase of power consumption because of the switch in host mode to process io request.Another reason is qemu iothread mutex contention.Threads blocking on average 20 us, when the iothread mutex is contended.

As in memory intensive benchmark experiment, we are able to observe reduction in power consumption and degradation of performance, but also an increase of the energy consumption.

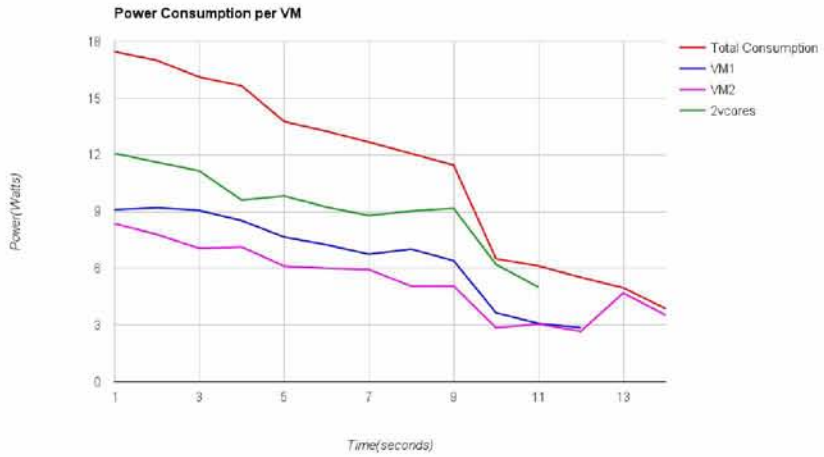

Figure 14: Power attribution (IO)

4.5 COMBINATION

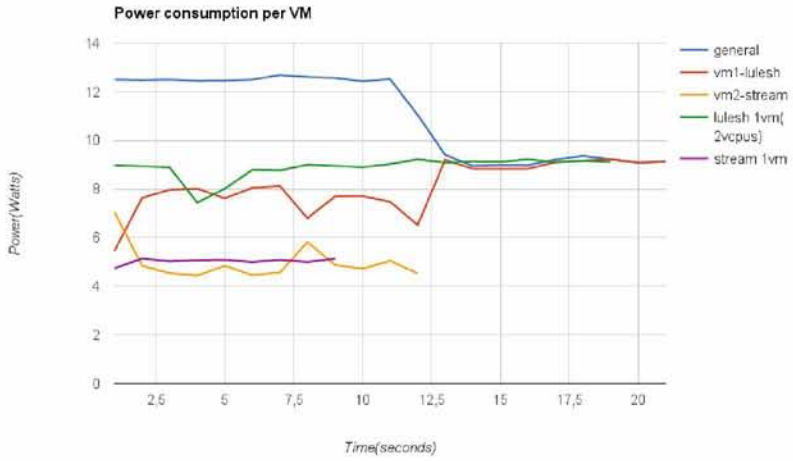

Figure 15: Cores power consumption

In Figure 15, we can see the power consumption extracted from rapl counters (general). The purple and the green line show the VM consumption that run the stream and lulesh respectively . Red and yellow lines depict the attributed power consumption of two running in parallel VM , while running stream and lulesh simultaneously. We observe that the individual consumption is properly approached.

5

RELATED WORK

Until today, there has been some progress on the issue of attributing energy consumption for running in parallel applications and specifically for virtual machines. Some examples are mentioned below.

Through the Host power Management of Vmware Vsphere we are able to come to a conclusion about the energy consumption of the virtual machines, and configure the system as well, in order to reduce the energy which is consumed in a cluster. However, Vsphere is a closed commercial application.

Intel VTune is another tool, by which we can do static analysis in applications or even in virtual machines. Through VTune we are able to draw conclusions about code's points, which are energy-intensive.

Apart from the model that we used in order to do the attribution of the energy consumption, other techniques have been suggested. One of them is, the [18] wherein consumption is approached, by grouping the threads of the same category. Another one is the [19] where power containers are implemented and provide an approach of the consumption in every core, including mechanisms that isolate the energy consumption of a multicore server at request level. Finally, energy consumption is approached in [1], considering the hyperthreading. In particular, it calculates the cpu-cycles, as they are calculated in our implementation. In addition,authors calculates the cycles as well, when one of the two virtual cores of a physical core is being used. Using the above in a mathematical model, it approaches accurately the energy consumption of each application.

6

CONCLUSION

In modern datacenter it is necessary to control the energy consumption. Administrators should be able to properly configure the host to a specific server of the datacenter to reduce consumption. Controlling virtualization stack demands of use different API, which for different hypervisor require different configuration or implementation. This problem is solved by using Libvirt that implements one API with different underlying implementation for various hypervisors.

In this thesis we developed a system for power attribution extracting counters of unhalted cycles and socket power consumption using perf events that will be enabled through Libvirt. Furthermore, we developed an application that uses the counters of these events in order to inform us about the individual consumption of virtual machines running in parallel on a server.The main goal was to describe completely the process of adding functionality to the library, in order to facilitate future development .

In the x86 architecture, energy attribution in individual applications is challenging. The reason is that the RAPL counters do not provide measurement of consumption per core. We used the model 1 to extract this measurments. The model we have used is sufficiently simple, but with the appropriate assumptions we can draw useful conclusions. Thus Intel should implement rapl counters per core in order to make the attribution easier. Finally based on the validation we did in Chapter 4, we observed that our approach to the problem of attribution of the power consumption is working properly and confirm everything we know about a type 1 hypervisor, ie lower performance overhead.

6.1 FUTURE WORK

From the measurements, we notice that the model we used, approximates correctly the energy consumption of a virtual machine. In order the model to be accurate, we have made the assumption that the virtual cpus of a virtual machine should be pinned to physical cores. Moreover, from model paper we can see that if we have hyperthreading enabled, the calculation of this simple model consumption can reach up to 40% deviation from the real consumption. Calculated by the model that takes into account the hyperthreading. This could be added to the existing system in the future.Another useful feature would be DRAM refresh rate scaling ,to help control system power consumption.

A

HOST CONFIGURATION

Listing A.1: virhostcpu.c

```
int virHostSetRaplLimit(int value,const char *type,const
    char *type_value)
{

    int fd;
    char file[128];
    const char *low = "50";
    const char *high = "100";


    if (value == 0){

            sprintf(file,
                "/sys/devices/system/cpu/intel_pstate/max_perf_pct");
            fd = open(file, O_RDWR);
            if(write(fd, low, 7) < 0){

                fprintf( stderr, "%s\n", strerror( errno ));
            }
    }
    else if(value == 1){

        sprintf(file,
            "/sys/devices/system/cpu/intel_pstate/max_perf_pct");
        fd = open(file, O_RDWR);
        if(write(fd, high, 7) < 0){

            fprintf( stderr, "%s\n", strerror( errno ));
        }
    }
    else{

        if(strcmp(type, "power_limit") == 0) {
            sprintf(file,
                "/sys/devices/virtual/powercap/intel-rapl/intel-rapl
            \\:0/intel-rapl\\:0\\:0/power_limit_uw");
            fd = open(file, O_RDWR);
```

```
            if(write(fd, type_value, 7) < 0){

                fprintf( stderr, "%s\n", strerror( errno ));
            }
        }
        else if(strcmp(type, "time_window") == 0) {
            sprintf(file,
                "/sys/devices/virtual/powercap/intel-rapl/intel-rapl
            \\:0/intel-rapl\\:0\\:0/time_window_us");
            fd = open(file, O_RDWR);
            if(write(fd, type_value, 7) < 0){

                fprintf( stderr, "%s\n", strerror( errno ));
            }
        }
        else if(strcmp(type, "enabled") == 0) {
            sprintf(file,
                "/sys/devices/virtual/powercap/intel-rapl/intel-rapl
            \\:0/intel-rapl\\:0\\:0/enabled");
            fd = open(file, O_RDWR);
            if(write(fd, type_value, 7) < 0){

                fprintf( stderr, "%s\n", strerror( errno ));
            }
        }
            return -1;
    }
    close(fd);
    return 0;
}


int virHostSetGovernor(int core,int governor)
{
    int fd;
    char file[128];
    const char *governor0 = "ondemand";
    const char *governor1 = "userspace";
    const char *governor2 = "performance";
    const char *governor3 = "powersave";


    sprintf(file, "/sys/devices/system/cpu/cpu%d/cpufreq/
                   scaling_governor", core);
    fd = open(file, O_RDWR);
    if(governor == 0){
        if(write(fd, governor0, strlen(governor0)) < 0){

            fprintf( stderr, "%s\n", strerror( errno ));
        }
    }
```

```
    else if (governor == 1){
        if(write(fd, governor1, strlen(governor1)) < 0){

            fprintf( stderr, "%s\n", strerror( errno ));
        }
    }
    else if (governor == 2){
        if(write(fd, governor1, strlen(governor2)) < 0){

            fprintf( stderr, "%s\n", strerror( errno ));
        }
    }
    else if (governor == 3){
        if(write(fd, governor1, strlen(governor3)) < 0){

            fprintf( stderr, "%s\n", strerror( errno ));
        }
    }
    else{

        printf("Wrong governor.\n");
        return -1;
    }
    close(fd);
    return 0;

}


int virHostSetFrequency(int core,const char *frequency)
{
    int fd;
    char file[128];

    sprintf(file, "/sys/devices/system/cpu/cpu%d
              /cpufreq/scaling_setspeed", core);
    fd = open(file, O_RDWR);
    if(write(fd, frequency, 7) < 0){

        fprintf( stderr, "%s\n", strerror( errno ));
    }
    close(fd);
    return 0;
}
```

BIBLIOGRAPHY

[1] S. Eranian Y. Zhai, X. Zhang. Happy: Hyperthread-aware power profiling dynamically. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, Philadelphia, PA, June 2014.

[2] Khoa Huynh, Stefan Hajnoczi. Kvm/qemu storage stack performance discussion, 2010. [Online; Retrieved January 3, 2015].

[3] Martin Dimitrov. Intel® power governor. `https://software.intel.com/sites/default/files/m/d/4/1/d/8/power_domains2.jpg/`, 2012.

[4] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. vol 3, 2016.

[5] Stefan Hajnoczi. Qemu internals: Overall architecture and threading model. http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html, 2011.

[6] Brad Ellison Lauri Minas. *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. Intel Press, 2009.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference*, pages 41–46, 2005.

[8] Libvirt Developers. Libvirt - the virtualization api. `http://libvirt.org/index.html`.

[9] Mike Murphy Patrick Mochel. sysfs - the filesystem for exporting kernel objects. `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`, 2011.

[10] S.Kuo P. Irelan. Performance monitoring unit sharing guide. `http://linux-security.cn/ebooks/ols2007/OLS2007-Proceedings-V1.pdf`.

[11] The performance monitoring interface for linux. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[12] Eric Blake. *Libvirt API extensions*. Redhat, 9 2010. https://libvirt.org/api_extension.html.

[13] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[14] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[15] W. Norcutt. Iozone benchmark program. `http://www.iozone.org/`, 1991.

[16] Q. Gao J. Che, Q. He and D. Huang. Performance measuring and comparing of virtual machine monitors. In *Embedded and Ubiquitous Computing*, page 381 –386, EUC '08. IEEE/IFIP International Conference on, December 2008.

[17] M. A. Murphy M. Fenn and S. Goasguen. A study of a kvm-based cluster for grid computing. In *Proceedings of the 47th Annual Southeast Regional Conference*, page 34:1–34:6, New York, NY, USA, 2009.

[18] B. Urgaonkar J. Choi, S. Govindan and A. Sivasubramaniam. Profiling, prediction, and capping of power consumption in consolidated environments. In *Proceedings of Modeling Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, pages 1–10, 2008.

[19] S. Dwarkadas X. Zhang K. Shen, A. Shriraman and C. Zhuan. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *Proceedings of 18th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, Texas, 2013.