Improving the Efficiency of Cloud Infrastructures with Elastic Tandem Machines

Frank Dürr
Institute of Parallel and Distributed Systems (IPVS)
Universität Stuttgart
Stuttgart, Germany
frank.duerr@ipvs.uni-stuttgart.de

***Abstract*—In this paper, we propose a concept for improving the energy efficiency and resource utilization of cloud infrastructures by combining the benefits of heterogeneous machine instances. The basic idea is to integrate low-power system on a chip (SoC) machines and high-power virtual machine instances into so-called Elastic Tandem Machine Instances (ETMI). The low-power machine serves low load and is always running to ensure the availability of the ETMI. When load rises, the ETMI scales up automatically by starting the high-power instance and handing over traffic to it. For the non-disruptive transition from low-power to high-power machines and vice versa, we present a handover mechanism based on software-defined networking technologies. Our evaluations show the applicability of low-power SoC machines to serve low load efficiently as well as the desired scalability properties of ETMIs.**

***Keywords*-cloud computing, infrastructure as a service, efficiency, energy, elasticity, scaling, system on a chip, software-defined networking, green computing**

I. Introduction

Cloud computing enjoys growing popularity because of its advantages like cost efficiency and virtually unlimited resources. In particular, the market of Infrastructure as a Service (IaaS) offers, where customers lease resources in form of virtual machines (VM), has expanded rapidly.

In this paper, we focus on the question how to improve the efficiency of IaaS compute infrastructures w.r.t. energy and utilization of physical resources. Today, efficiency is improved by two concepts: hardware consolidation through virtualization, and overbooking of physical resources. With *hardware consolidation*, multiple physical machines are replaced by a set of VMs hosted on few(er) physical hosts, assuming that customers demand VMs with lesser resources than offered by a physical host. This alleviates the problem of underutilized physical hardware. However, hardware consolidation alone does not ensure perfect utilization of physical resources if VMs do not fully use their nominal (maximum) resources permanently. Therefore, operators *overbook physical resources* by placing more VMs on a physical host than would be allowed by the sum of nominal VM resources [1], [2], [3].

Although we think that these are effective mechanisms in general, we argue that they are difficult to apply in scenarios where VMs often serve low load, but still need to be constantly available and able to provide their maximum nominal resources when load rises. A typical example could be a web or application server serving few requests spread over a larger time span with higher load only during "peak hours". Why is is hard to allocate such VMs efficiently, i.e., without wasting (idle) resources and energy? According to the consolidation and overbooking principles described above, one would host several VMs on one physical host. According to the considered scenarios, many of these VMs are only weakly loaded and possibly idle most of the time. Therefore, there are several options to achieve efficiency:

First, one could *increase the factor of overbooking* to a large value to fully utilize the physical resources of hosts. However, increasing this factor also increases the risk of overloading hosts when too many VMs request their nominal resources. To ensure the service level agreement (nominal VM resources), busy VMs have to be migrated to other hosts. Although this is a valid approach, it requires complex models and adaptation mechanisms. The aggregated statistical resource demand of VMs has to be estimated, including processing and network resources, and suitable sets of VMs must be identified for optimal VM placement [1], [2], [3]. To minimize the downtime of VMs, live migration mechanisms are required whose overhead must be taken into account [4].

Secondly, *idle VMs could be switched off or swapped out to disk* and booted or restored on demand. (Note that cloning a running VM is not an option here since this requires a running VM itself.) However, this contradicts the availability requirements of many services. For instance, consider a web service. If a packet of an HTTP request arrives at the border router of the datacenter, it takes at maximum a few milliseconds until the datacenter network has forwarded this packet to the host. Delaying the connection request artificially is no option since requests have to be served within few hundred milliseconds or even tens of milliseconds for typical web services. However, existing boot concepts require several seconds up to minutes to start a VM [5].

Thus, increasing the factor of overbooking is risky and requires complex adaptation mechanisms, and switching off VMs is not an option because of availability constraints. Therefore, we try to avoid these difficulties by proposing a radically different approach in this paper: *We utilize heterogeneous physical hardware* optimized for different scenarios, namely low load, performance, and energy consumption vs. high load, performance, and energy consumption. Our goal can be stated as follows: *We try to implement the ideal machine w.r.t. efficiency that consumes no power while being idle or weakly loaded—so it can be always on and, thus, be available when requests arrive—, and that scales up to the requested nominal resources of the VM (elasticity).*

Unfortunately, such an ideal machine does not exist. However, with respect to very low energy consumption, the current trend of system on a chip (SoC) computers comes very close with at least low power consumption while being idle and in fact also being active. For instance, the Raspberry Pi as a prominent example only consumes few Watts and fulfills the essential hardware requirements of low-performance servers—comparable, for instance, to Amazon's Micro Instance—, featuring a 700 MHz CPU, 512 MB RAM, and 100 Mbps network interface. Therefore, we advocate to utilize such SoC computers to implement *Low-power Micro Instances (LPMI)* in cloud datacenters.

Beyond proposing to use such low-power SoC computers as LPMIs, we also tackle the major problem of these machines, namely, their limited performance. To this end, we propose an approach to adaptively scale up LPMIs to the nominal requested resources. The basic idea is as follows: As long as there are only few requests, we use an LPMI that is never switched off. If the load increases beyond the performance limits of the LPMI, we boot a more powerful (traditional) VM with the nominal resources and forward requests to this *High-power Instance (HPI)*. We will present a handover protocol based on state of the art software-defined networking (SDN) technologies to make this transition from LPMI to HPI and vice versa *non-disruptive* and *transparent* to the client. That is, no requests are lost during the transition, and to the client the combination of LPMI and HPI looks like an (almost ideal) elastic machine (called *Elastic Tandem Machine Instance*; *ETMI*) accessible through one public IP address and scaling from zero to the nominal resources.

In detail, we make the following contributions: (1) Performance evaluations of low-power SoC machines in realistic three-tier application settings showing the applicability of LPMIs for middle tier services with low load. (2) An architecture and concept for combining LPMIs and HPIs into efficient and elastic ETMIs. (3) A mechanism for the seamless handover between LPMIs and HPIs using SDN technologies. (4) A proof-of-concept implementation and evaluation of the effectiveness and energy-efficiency of the approach.

The rest of this paper is structured as follows. First, we present our system model in Sec. II. In Sec. III, we present the architecture, concept, and handover protocol to implement ETMIs. In Sec. IV, we evaluate the performance and efficiency of LPMIs based on SoC hardware and show the scalability of ETMIs. Finally, we give an overview of related work in Sec. V, and conclude the paper in Sec. VI.

II. System Model

In this section, we present the target environment and assumptions for our approach. We consider a single datacenter of an IaaS provider with physical machines (*hosts*) made of commodity hardware hosting VMs. We call VMs running on these hosts *High-power Instances (HPI)* since they offer more resources and higher performance than our low-power instances running on SoC hardware. Moreover, they consume more energy. The nominal rating of HPIs varies according to the demands of the customer. Here, we just assume that HPIs offer significantly more performance than low-power instances.

In addition, the datacenter also contains hosts based on low-power SoC hardware hosting *Low-power Micro Instances (LPMI)*. In contrast to HPIs, LPMIs are not virtualized, because the type of SoC hosts that we consider only has resources for hosting one LPMI (therefore, a SoC host is equivalent to an LPMI in the following). We assume that these LPMIs consume a very small amount of energy, typically only few Watts. They have fewer computational, memory, and network resources than an HPI, however, still sufficient for running a web or application server at low load (cf. Sec. IV for a backup of this assumption). As a "blueprint" for a SoC-based LPMI, we use the Raspberry Pi, with 700 MHz CPU, 512 MB RAM, and 100 Mbps NIC.

Both, HPIs and LPMIs are connected to the same *datacenter network* consisting of *switches*, which provide connectivity between machines and via routers to the Internet (cf. Fig. 1). We assume that at least the *core switches* providing connectivity to clients outside the datacenter, implement the OpenFlow standard for SDN [6]. In plain words, OpenFlow enables a *controller* process running on a server to configure the forwarding tables of switches. Typically, multi-layer switches are used, which can make forwarding decisions based on layer 2–4 header fields (e.g., source and destination MAC or IP addresses, and port numbers). The OpenFlow protocol is already supported by several switches of major vendors. Therefore, assuming OpenFlow core switches to be available is realistic. Note that it is one feature of our approach that only core switches need to support OpenFlow. We assume that all other switches perform layer 2 forwarding using MAC addresses and virtual LAN ids.

With respect to the hosted application systems, we target systems based on a typical *three-tier architecture. Our approach for implementing efficient machine instances is targeted at services from the middle tier* such as web and application servers. These servers are executed on HPIs or LPMIs. We assume that all persistent data and state information is stored by backend services such as databases or file servers executed on dedicated machines that are always on and beyond the scope of our optimization. Moreover, we assume that middle tier services only use volatile state information that is only relevant for individual requests; all other information is stored in the backend or client, e.g, as HTTP cookies. For instance, a web server using PHP or a servlet engine might query information from a database, process it, and write the results back to the database within one request from a web client. As we will see later, this assumption is important since we cannot transfer state information easily from an LPMI to an HPI using VM migration. The reason for this assumption is that SoC hosts not necessarily use the same hardware platform as the high-power hosts. For instance, high-power hosts are often based on x86 architectures, whereas the Raspberry Pi SoC computer is using an ARM platform. Although emulating an ARM platform on an x86 machine is possible, it induces a performance penalty and is therefore not considered further. Moreover, we want to avoid the overhead and difficulties of migration. Looking at typical web-based applications, e.g., LAMP systems (Linux, Apache, MySQL, PHP) or servlet engines like Google AppEngine connected to a persistent data storage (e.g., Google BigTable), this assumption holds true for a large set of systems.

III. Approach for Implementing Scalable and Efficient Machine Instances

In this section, we present our approach for implementing efficient and scalable machine instances. We first present the basic idea and architecture of our system, before presenting details about the essential algorithm of our approach: a non-disruptive handover protocol based on SDN technologies to switch adaptively between LPMIs and HPIs.

A. Overview

As stated, our goal is to offer an *Elastic Tandem Machine Instance (ETMI)* that is always available and scaling up to the requested nominal resources. We achieve this by combining one LPMI and one HPI into an ETMI. We assume that the LPMI and HPI are installed with the same middle tier service software with identical configuration, e.g., web servers referencing the same document root directory on a remote file server, or servlet engines using the same remote database. Therefore, requests to both instances behave identical, besides the different performance of LPMI and HPI.

To leverage the benefits of SoC hardware, only the LPMI is running in low load situations to save energy. If the load rises beyond the limits of the LPMI, an HPI is booted automatically and further requests are forwarded to the HPI (scale up). During the transition—i.e., as long as the HPI is booting—requests are still forwarded to the LPMI, i.e., the ETMI is constantly available (therefore, this could also be termed a fastboot concept for HPIs). If the load of the HPI drops below a certain threshold, it is switched off and requests are forwarded to the LPMI again (scale down).

Adaptive forwarding of traffic from clients to the LPMI or HPI is performed "in hardware" in the communication network by configuring core switches. Therefore, the throughput and delay (response time) do not suffer. The control logic is implemented in software by the SDN controller.

To the client outside the datacenter, this process should be transparent, i.e., the ETMI looks like a single machine instance accessible through one public IP address without any disruption during handover. The major difficulty is to implement a handover mechanism where *already established* network connections from clients to the LPMI or HPI do not break during the transition (for instance, HTTP 1.1 sends multiple HTTP requests over the same TCP connection). A TCP connection includes state information such as connection state, segment counters, or buffers. This information is kept in the kernel space. Since we cannot easily use VM migration, we have to make sure that already established connections are still served by the original instance that accepted the connection initially until they are closed regularly. In Sec. III-C, we will present a suitable handover protocol, after we have introduced the basic architecture of our approach in the next subsection.

B. Architecture of Elastic Tandem Machine Instances

Figure 1 shows the architecture of our system including its components running in one datacenter. First of all, an *LPMI* and an *HPI* are required to implement one *ETMI*. The LPMI and HPI of an ETMI do not need to be co-located in one rack. In addition to LPMIs and HPIs hosting middle tier services, *backend servers* like database or file servers are required for persistent data storage.

The core component of our approach is the *SDN controller*, which is responsible for directing packets from clients outside the datacenter to either the LPMI or HPI, depending on the load. To this end, it configures the forwarding table of the *core switches* using the OpenFlow protocol. All core switches are configured with the same forwarding table entries. The protocol for configuring these switches is described in Sec. III-C. Besides the core switches, the datacenter network consists of more switches like aggregation, top-of-rack, and virtual switches to connect VMs on hosts. These switches just perform standard layer-2 forwarding.

Both LPMI and HPI have individual (possibly private) IP addresses assigned to their network interfaces, denoted as a_{LPMI} and a_{HPI}, respectively. Additionally, we assign the public IP address of the service, say a_{public}, to the same network interface using IP aliasing (with IP aliasing, multiple IP addresses can be assigned to the same NIC). The basic ideas is to use individual addresses for the internal communication with the controller. Messages from clients to the public IP address are directed to either the LPMI or HPI by the core switch using MAC address rewriting configured by the controller (see Sec. III-C).

Another component to control the adaptation process is the *load monitor* to determine the system load of instances. Individual load monitors are executed on the LPMI and

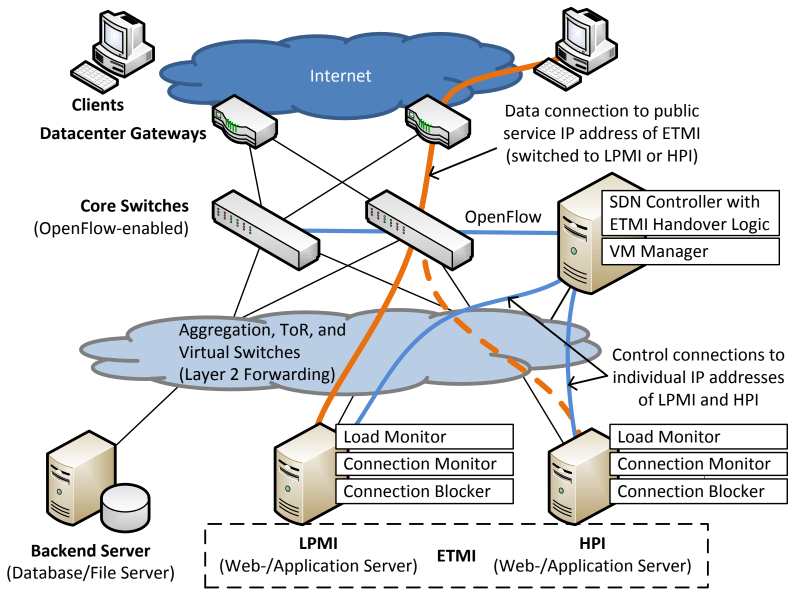

Internet
Clients
Datacenter Gateways
Data connection to public service IP address of ETMI (switched to LPMI or HPI)
Core Switches (OpenFlow-enabled)
OpenFlow
SDN Controller with ETMI Handover Logic
VM Manager
Aggregation, ToR, and Virtual Switches (Layer 2 Forwarding)
Control connections to individual IP addresses of LPMI and HPI
Load Monitor
Connection Monitor
Connection Blocker
Load Monitor
Connection Monitor
Connection Blocker
Backend Server (Database/File Server)
LPMI (Web-/Application Server)
ETMI
HPI (Web-/Application Server)

Figure 1. System architecture (connections between the backend server and LPMI/HPI are not depicted)

the HPI to notify the controller of LPMI overload or HPI underload. Since our focus is not on load monitoring or prediction, we use simple load metrics which can be easily measured on instances (CPU run queue length and request (data) rate; cf. Sec. IV). For these metrics, we define threshold values to define the overload of the LPMI (T_{overload}) and underload of the HPI ($T_{\text{underload}}$) including a hysteresis to prevent oscillation. Moreover, we inhibit switches from HPI to LPMI during a pre-configured interval $\Delta t_{\text{inhibit}}$ ($120\,\text{s}$ in our experiments) after a switch from LPMI to HPI to avoid rapid shutdowns and reboots of the HPI.

In case of an overloaded LPMI, the controller starts an HPI via the *Virtual Machine Manager*, which notifies the controller when the HPI is running.

As will become clear from the description of the handover protocol, we need two more components to support the adaptation process. First, *connection monitors* are executed on the LPMI and HPI. On a request from the controller, these monitors determine all established TCP connections from clients to the instance and send their information (source port, source IP address, destination port, destination IP address) to the controller. Secondly, *connection request blockers* on the LPMI and HPI block new connection requests from clients to the instance during handover. Next, we present the handover protocol in detail.

C. SDN-based Handover Protocol

Whenever the LPMI becomes overloaded or the HPI becomes underloaded, the controller executes a handover protocol to redirect further requests to the other instance. To direct traffic to either the LPMI or HPI, we use MAC address rewriting performed by the core switches based on forwarding table entries installed by the controller. To direct packets to the LPMI, an action is installed for incoming packets matching the destination IP address a_{public} that rewrites the destination MAC address to the MAC address of the LPMI, say m_{LPMI}. Similarly, packets can be directed to the HPI by rewriting the destination MAC address to the MAC address of the HPI m_{LPMI}. Note that after rewriting, the datacenter network performs layer 2 forwarding to the LPMI and HPI based on the MAC destination address rather than using the destination IP address, and instances will deliver the packet since their NIC is configured with a_{public} using IP aliasing. Also note that outgoing packets to the client do not need any special actions, and clients always receive packets from the source address a_{public}.

Next, we explain the handover protocol using a typical sequence of operations during the handover process (cf. Fig. 2). Assume that initially load is low and only the LPMI is running; all requests are forwarded to the LPMI. Then, the load is increasing. If the load of the LPMI is above the overload threshold, the LPMI's load monitor sends an *overload notification message* to the controller triggering the handover process with the following steps:

Step 1—Booting HPI: The controller requests an HPI to be started by the VM Manager (line 5). When the HPI has been started, the controller receives a *boot complete message*.

Step 2—Pinning established connections: The controller makes sure that already established TCP connections will stay connected to the LPMI to prevent broken connections after redirecting traffic to the HPI. To this end, the controller first queries the connection monitor of the LPMI for all established connections (TCP state "established") or connection requests that have already been acknowledged

```
 1: procedure onOverloadNotificationFromLPMI
 2:     if $t_{\text{now}} - t_{\text{lastswitch}} < \Delta t_{\text{inhibit}}$ then
 3:         return;                              ▷ ignore notification
 4:     end if
 5:     VMManger.bootAndWaitForHPI();
 6:     ConnectionBlockerLPMI.blockSynRequests();
 7:     $C \leftarrow$ ConnectionMonitorLPMI.getConnections();
 8:     for all $c \in C$ do             ▷ pin established connections
 9:         addFwdTableEntry:
10:             match$\{c.a_{\text{src}}, c.a_{\text{dest}}, c.port_{\text{src}}, c.port_{\text{dst}}\} \rightarrow$
11:             action$\{dstMac = m_{\text{LPMI}}\}$;
12:     end for
13:     modifyFwdTableEntry:             ▷ fwd new conn. to HPI
14:         match$\{a_{\text{public}}\} \rightarrow$ action$\{dstMac = m_{\text{HPI}}\}$;
15:     ConnectionBlockerLPMI.unblockSynRequests();
16:     $t_{\text{lastswitch}} \leftarrow t_{\text{now}}$;
17: end procedure
```

Figure 2. Handover control logic: switch from LPMI to HPI

by the LPMI (TCP state "syn recvd") (Step 2.1; line 7). For each connection, the controller receives the four tuple (source and destination IP addresses and port numbers) uniquely identifying a TCP connection. For each tuple, the controller installs a forwarding table entry in the core switches (Step 2.2; line 8–12) matching the complete four tuple and rewriting the MAC address to m_{LPMI}. These forwarding table entries are assigned a higher priority than the entry of Step 3, so it is guaranteed that packets of established connections are treated according to these rules.

Step 3—Redirecting new connections: All packets to a_{public} that are not part of an established connection are redirected to the HPI. To this end, the controller installs another forwarding table entry matching a_{public} to rewrite the destination MAC address to m_{HPI} (line 13–14). This forwarding table entry is assigned a lower priority than the more specific entries for individual connections from Step 2.2, so it will only affect packets of new connections.

So far, this protocol suffers from a race condition (cf. Fig. 3). Packets of connections established to the LPMI *after* querying open connections (Step 2.1) but *before* installing forwarding table entries for redirecting packets to the HPI (Step 3) will also be redirected to the HPI since they will not be pinned. This would lead to broken connections since these packets should go to the LPMI, which accepted the connection initially. Although the time between these steps might be very small (milliseconds), we need to deal with this case using the following mechanism.

We prevent establishing new connections to the LPMI between the beginning of Step 2.1 and end of Step 3. To this end, the LPMI drops all packets with a SYN bit set in the period before querying established connections until the redirection has been installed. We achieved this by configuring the firewall of the LPMI through the connection

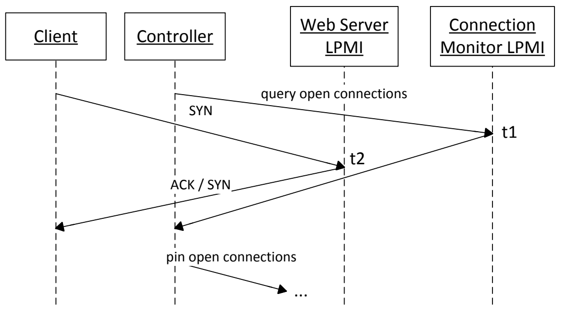


Figure 3. Race condition: the connection accepted at time t_2 is not included in connections queried at t_1 and, thus, will not be pinned.

blocker component (line 6, 15). Since IP packets can get lost on their route from clients to the server anyway (incl. TCP SYN requests), a client would simply repeat its connection request after a small timeout, and then connect to the HPI.

The handover process from the HPI to the LPMI is symmetric with two exceptions: (1) The LPMI does not need to be booted since it is always running. (2) The HPI is shut down down, after the last established connection to the HPI has been closed by the client. Since long-lived connections to the HPI could prevent the HPI from shutting down, the connection idle timeout value of servers should be set to smaller values, e.g., 60 s. Additionally, underloaded HTTP servers can close connections after the next request gracefully using the HTTP "connection close" directive (this requires a modification of the server software).

A last thing to mention is that the forwarding table entries from Step 2 can be removed by the controller as soon as the established connections are closed to avoid an ever growing forwarding table whose space is limited by the switch memory. To this end, the controller periodically queries the LPMI and HPI for established connections and removes vanished connections from the forwarding table of the core switches.

IV. Evaluation

In this section, we first backup our claim that LPMIs using low-power SoC hardware have sufficient performance to serve low load. To this end, we evaluate LPMI performance using two realistic settings: a web server serving static web pages and a web server creating dynamic web pages. Secondly, we validate our concept of ETMIs using a proof-of-concept implementation to evaluate the scaling properties and non-disruptive handover mechanism. Thirdly, we evaluate the energy efficiency of LPMIs compared to traditional VMs on shared hosts.

A. LPMI Performance: Static Content

The first scenario for evaluating the performance of LPMIs is a web server delivering static web pages to clients. In this and all other evaluations, the LPMI is running on a Raspberry Pi SoC machine with 700 MHz CPU, 512 MB RAM, and 100 Mbps Ethernet NIC. The LPMI is installed with a Linux operating system and executes an Apache web server (middle tier service). Web pages are stored on an NFS file server (backend service) executed on a dedicated machine (Intel i5 quad-core with 2.67 GHz, 12 GB RAM, 1 Gbps Ethernet). These web pages were taken from a real website (http://www.netsys2013.de/) consisting of 43 web pages containing smaller images of average size 11 kB.

In all of our experiments, load (requests) is generated by clients according to a Poisson distribution $P_\lambda(k) = \frac{\lambda^k}{k!}e^{-\lambda}$ with λ requests per second on average. One request downloads a complete web page including stylesheets, images, etc.—thus, one request corresponds to a number of HTTP requests—selected randomly from the set of web pages. We start with a request rate of $\lambda_{\min} = 1$ request/s at $t = 0$ s and increase the rate every 50 s by 1 request/s up to $\lambda_{\max} = 30$ request/s.

As performance metric, we measure the response time per request (time to deliver a complete web page) and throughput (answered requests per second delivering a complete web page). Figure 4(a) shows the throughput and response time of the LPMI over time averaged over 10 runs. For the throughput, we see that the LPMI can serve requests up to 26 request/s. At this rate, the network connection of the LPMI becomes the bottleneck. The response time increases significantly at about 1000 s (20 request/s), which is an indication that the LPMI becomes overloaded. At this time, the response time exceeds 150 ms, which we also consider to be an upper limit for delivering web pages. Therefore, the processor of the LPMI is the most significant bottleneck in this scenario limiting the throughput to 20 request/s for the given maximum response time limit of 150 ms.

B. LPMI Performance: Dynamic Content

Besides static web pages, we also evaluated the creation of dynamic web content using a typical LAMP system setup: **A**pache web server executing a **P**HP script on the LPMI; **M**ySQL database on backend server. The script queries a database with 10,000 bank accounts to execute a transaction increasing the value of a randomly selected account. A web page is created showing the new value of the account.

Figure 4(b) shows the resulting throughput and response time over time for 1 request/s $\leq \lambda \leq$ 80 request/s. We see that the LPMI achieves a higher throughput in this experiment up to 78 request/s since the network is less loaded for small SQL queries and the delivered smaller web pages. The response time starts to increase significantly and exceeds the limit of 150 ms at about 3600 s (70 request/s), again due to a processor bottleneck.

As most demanding scenario w.r.t. processing, we evaluated a Java servlet engine (Tomcat) executed on an LPMI. We implemented a servlet extracting and returning all speeches of Hamlet from an XML document with all speeches of Shakespeare's tragedy Hamlet (288 kB). In this scenario, the LPMI becomes overloaded already at 0.5 request/s when the response time starts to increase significantly. The minimum response time is about 1.1 s. Therefore, we consider such processing-intensive services to be too demanding for low-power SoC machines as long as the request rate is not very low and high response times can be tolerated by the application.

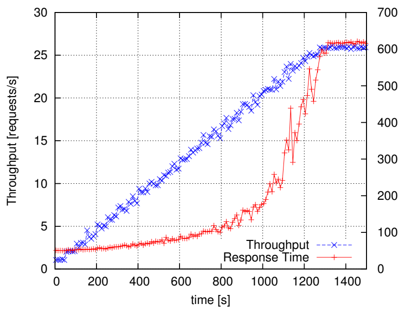
C. ETMI Performance

The evaluations presented so far have shown that LPMIs alone could be used as replacement for virtual micro instances in realistic scenarios with moderate processing demands and load. However, they also have shown their performance limitations for higher load. Next, we show that ETMIs can overcome these limitations of LPMIs. To show the effectiveness of ETMIs, we implemented a prototype performing the handover protocol from Sec. III.

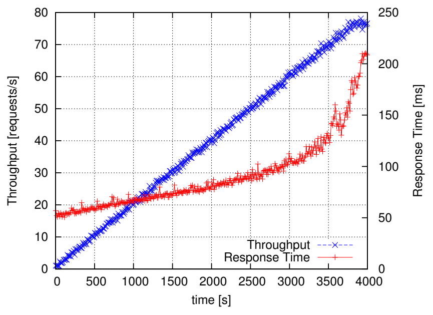
In these experiments, we use the same LPMI as in the previous experiments. The HPI has two CPU cores rated at 4.2 GHz, 2 GB RAM, and a 1 Gbps NIC. The core switch is an OpenFlow software switch (Open vSwitch) using two 1 Gbps ports (one upstream and one downstream port). This switch has sufficient performance to forward traffic at line rate (1 Gbps in each direction) and, therefore, does not become a bottleneck in our experiments using only one ETMI. The rest of the network (aggregation switches, ToR switches, etc.) is emulated by one 1 Gbps layer-2 hardware switch connected to the core switch, LPMI, HPI, controller, and backend server. The controller is implemented as extension to the Floodlight OpenFlow controller using its REST (northbound) interface to setup flow table entries. The connection monitors of LPMI and HPI use the socket statistics command (`ss`) to determine established network connections. The connection blocker is implemented using firewalls on the LPMI and HPI.

During our experiments, we found out that load monitoring to detect over- and underload is far from trivial. First, we used the exponentially smoothed CPU run queue length (RQL) as reported by the `proc` file system on LPMI and HPI. However, if the HPI has much higher performance than the LPMI (as in our experiments), the RQL metric is problematic since load close to the overload threshold of the LPMI leads to very low load (close to zero) of the HPI. Therefore, after the inhibit time, the HPI will report an underloaded system, and the controller switches back to the LPMI, which becomes overloaded immediately again provoking a switch to the HPI, and so forth.

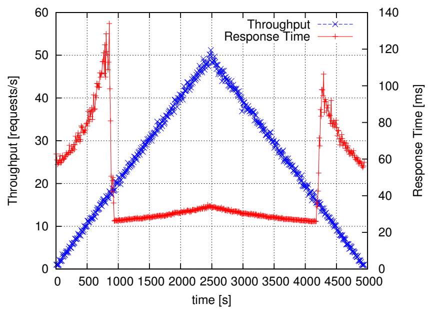
Ideally, load should be described by a metric that is easy to measure by the LPMI and HPI, and whose values can be used as indicator for the performance of the *other* instance of the ETMI under the current load. Then the switch from HPI to LPMI only takes place if the metric indicates that the LPMI will not become overloaded by the current load. We decided to use the data rate of requests as metric since



(a) LPMI: static web pages



(b) LPMI: dynamic web pages



(c) ETMI: static web pages

Figure 4. Throughput and response time

for many services it correlates well with the LPMI system load (a higher volume of incoming data usually means more requests or data to process). Alternatively, we could also instrument the web or application servers to directly measure the number of request on the application level. Based on offline benchmarks, we set $T_{overload} = 80\,\mathrm{kB/s}$ and $T_{\mathrm{underload}} = 53\,\mathrm{kB/s}$ as LPMI overload and HPI underload threshold, respectively. The overload threshold includes a "safety margin" to switch before the limit of the LPMI is reached. We measured the data rate of incoming requests to the public IP address using firewall rules on LPMI and HPI.

As scenario, we use the web server from Sec. IV-A serving static web pages. Web servers with the same configuration are executed on LPMI and HPI, both fetching web pages from the same NFS file server. Again, we use a Poisson process with $\lambda_{\min} = 1\,\mathrm{request/s}$ and $\lambda_{\max} = 50\,\mathrm{request/s}$, i.e., far beyond the identified limits of an LPMI. When the rate reaches $\lambda_{\max}$, we decrease it again down to $\lambda_{\min}$ by $1\,\mathrm{request/s}$ every $50\,\mathrm{s}$.

Figure 4(c) shows the throughput and response time of the ETMI over time. At $t = 900\,\mathrm{s}$ ($18\,\mathrm{request/s}$), the controller switched from the LPMI to the HPI; at $t = 4200\,\mathrm{s}$, it switched back to the LPMI. We verified that during the transitions, no established HTTP connections broke in all runs. The throughput now shows an ideal behavior rising up to the maximum request rate of $50\,\mathrm{request/s}$ and then decreasing again according to the decreasing request rate. This shows the desired ability to scale ETMIs up and down adaptively with a seamless transition.

The response time now always stays below the limit of $150\,\mathrm{ms}$. As we can see, the response time is higher when the LPMI serves the load at the beginning and end of the experiment. However, before it rises above the critical limit of $150\,\mathrm{ms}$, the load monitor detects an overloaded system and switches to the HPI. Moreover, the hysteresis successfully prevents the LPMI from becoming overloaded after the switch from HPI to LPMI. Overall, this experiments shows the desired scaling behavior of the ETMI.

D. *Energy Efficiency*

One goal of ETMIs is to improve the energy efficiency in low load situations when the ETMI is running on SoC hardware only, i.e., as long as the LPMI is serving load. Therefore, we compare the energy efficiency of LPMIs (Raspberry Pi) to VMs running on shared physical hosts (AMD Athlon 64 X2 Dual Core $4.2\,\mathrm{GHz}$).

While being idle, we measured a power consumption of $P_{\mathrm{idle,LPMI}} = 1.85\,W$ for the LPMI and $P_{\mathrm{idle,host}} = 141.22\,\mathrm{W}$ for the VM host. Thus, $P_{\mathrm{idle,host}}/P_{\mathrm{idle,LPMI}} \approx 76$ VMs have to be placed on the host to achieve the same idle energy efficiency as running 76 LPMIs.

For a load of $4\,\mathrm{request/s}$ per LPMI (static web page scenario of Sec. IV-A), we measured a power consumption of $76 \times 1.89\,\mathrm{W} = 143.64\,\mathrm{W}$ for 76 LPMIs and $184.46\,\mathrm{W}$ for a host serving an equivalent rate of $76 \times 4\,\mathrm{request/s} = 304\,\mathrm{request/s}$. Thus, 76 VMs consume about 1.28 times more energy than 76 LPMIs.

The gain in energy efficiency increases drastically, when a single shared physical host cannot server the same load anymore as an equivalent number of LPMIs, and VMs have to be distributed to multiple physical hosts. In our experiments, one physical host can serve a maximum of about $300\,\mathrm{request/s}$. Therefore, at $8\,\mathrm{request/s}$ per LPMI, two physical hosts are necessary to serve the same load ($608,\mathrm{request/s}$) as 76 LPMIs. At $8\,\mathrm{request/s}$ per LPMI, we measured a power consumption of $76 \times 1.92\,\mathrm{W} = 145.92\,\mathrm{W}$ for 76 LPMIs. Two physical hosts serving the same load consume $2 \times 184.46\,\mathrm{W} = 368.92\,\mathrm{W}$. Thus, 76 VMs running on two shared physical hosts consumed 2.53 times more power than a performance-equivalent number of LPMIs.

V. Related Work

In this section, we give an overview of related approaches. As already described in Sec. I, approaches based on resource overbooking such as [1], [2], [3] target the same goals as our approach, namely, to increase energy efficiency and utilization of physical resources. Due to the problems of heavy overbooking (risk of overloading) and on-demand booting or restoring of VMs (reduced availability due to latencies), we investigated a different approach in this paper utilizing heterogeneous hardware to efficiently support idle and weakly loaded VMs without sacrificing availability.

In [7], the authors show that integrating heterogeneous hardware (here Intel Atom SoC and Xeon systems) can be used to achieve energy-proportional systems. As one design option, they identify the integration of discrete server systems. Our hand-over approach proposed in this paper is a network-centric mechanism to enable this integration.

From a technical point of view, our approach shares similarities with load balancing mechanisms. A common principle is to first forwarded requests to a load balancer, which then directs the requests to machines from a pool of servers. Similar to our approach, this redirection can be performed by rewriting addresses, either IP addresses (network address translation (NAT), e.g., [8]) or MAC addresses (e.g., [9]). In contrast to most of these approaches, we avoid a dedicated load balancer and rather utilize available (core) switches together with an SDN controller to implement request redirection.

A load balancing approach also utilizing SDN was proposed in [8]. Although this approach is based on NAT instead of MAC address rewriting, the problem of keeping established TCP connections alive is the same as in our approach. The solution proposed in [8] has two drawbacks: Either it redirects packets to the controller to distinguish between old and new connections, which might lead to a bottleneck at the (software) controller. Or it uses a heuristic (60 s timeout) to detect closed connections on the switch. This heuristic might either lead to broken connections if packets are sent *after* 60 s of inactivity. Or it might even prevent the handover since also new requests are forwarded to the old instance during the timeout period, which restarts the timer. We avoid these problems using readily available connection state from the end systems (HPI, LPMI), a feature enabled by the application-level SDN controller.

VI. Summary

In this paper, we presented a concept to implement Elastic Tandem Machine Instances, which combine the benefits of low-power SoC hardware (low energy consumption) and high-power virtual machine instances (large resources). While the low-power instance serves low load and ensures constant availability, the high-power instance serves high load. We presented a concept to scale up ETMIs by switching adaptively from the low-power instance to the high-power instance. Using a handover protocol based on software-defined networking technologies, the transition is made seamlessly without disrupting existing connections. Moreover, we demonstrated the applicability of low-power SoC hardware to serve low load in realistic three-tier system settings, as well as a proof of concept of ETMIs.

As part of future work, we plan to improve our concept. First, we want to include concepts for self-tuning the threshold values based on user-defined quality of service parameters like minimum response time instead of using manually configured values based on offline benchmarking. Secondly, models to predict the load and performance can help us to better plan the startup and shutdown of high-power instances to reduce the time until the HPI is available. Thirdly, we are going to integrate more than two machines with different performance specifications through our hand-over protocol. This allows for scaling up seamlessly from a low-power SoC machine, to small, medium, and large instance virtual machines.

References

[1] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, “Efficient resource provisioning in compute clouds via VM multiplexing,” in *Proc. of the 7th International Conference on Autonomic Computing (ICAC ’10)*, Washington, DC, Jun. 2010, pp. 11–20.

[2] B. Uranokar, P. Shenoy, and T. Roscoe, “Resource overbooking and application profiling in shared hosting platforms,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI Winter 2002, pp. 239–254, Dec. 2002.

[3] D. Breitgand and A. Epstein, “Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds,” IBM Haifa Research Laboratory, Tech. Rep. H-0312 (H1201-006), Jan. 2012.

[4] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, “Cost of virtual machine live migration in clouds: A performance evaluation,” in *Proceedings of the 1st International Conference on Cloud Computing (CloudCom ’09)*, Beijing, China, Dec. 2009, pp. 254–265.

[5] M. Mao and M. Humphrey, “A performance study on the vm startup time in the cloud,” in *Proc. of the 5th IEEE International Coonference on Cloud Computing (Cloud ’12)*, Honolulu, Hawaii, Jun. 2012, pp. 423–430.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling innovations in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[7] B.-G. Chun, G. Iannaccone, G. Iannaccone, R. Katz, G. Lee, and L. Niccolini, “An energy case for hybrid datacenters,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 76–80, Jan. 2010.

[8] R. Wang, D. Butnariu, and J. Rexford, “OpenFlow-based server load balancing gone wild,” in *Proc. of the 11th Workshop on Hot Topis in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE ’11)*, Boston, MA, Mar. 2011, pp. 12–17.

[9] G. Goldszmidt and G. Hunt, “NetDispatcher: A TCP connection router,” IBM T.J. Watson Research Center, Tech. Rep. RC 20853, May 1997.