netShip: A Networked Virtual Platform for Large-Scale Heterogeneous Distributed Embedded Systems

YoungHoon Jung
Dept. of Computer Science
Columbia University
New York, NY 10027
jung@cs.columbia.edu

Jinhyung Park
The Fancy Thing Daemon, Inc.
New York, NY 10014
jp@thefancy.com

Michele Petracca
Cadence Design Systems, Inc.
San Jose, CA 95134
petracca@cadence.com

Luca P. Carloni
Dept. of Computer Science
Columbia University
New York, NY 10027
luca@cs.columbia.edu

ABSTRACT

From a single SoC to a network of embedded devices communicating with a backend cloud-computing server, emerging classes of embedded systems feature an increasing number of heterogeneous components that operate concurrently in a distributed environment. As the scale and complexity of these systems continues to grow, there is a critical need for scalable and efficient simulators. We propose a networked virtual platform as a scalable environment for modeling and simulation. The goal is to support the development and optimization of embedded computing applications by handling heterogeneity at the chip, node, and network level. To illustrate the properties of our approach, we present two very different case studies: the design of an Open MPI scheduler for a heterogeneous distributed embedded system and the development of an application for crowd estimation through the analysis of pictures uploaded from mobile phones.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: distributed systems, realtime systems and embedded systems

General Terms

Distributed Embedded Systems Design

Keywords

Android, Embedded Systems, MPI, OpenCV, OVP, QEMU, Simulation, System Design, Virtual Platform

1. INTRODUCTION

Computing systems are becoming increasingly more concurrent, heterogeneous, and interconnected. This trend happens at all scales: from multi-core systems-on-chip (SoC), which host a variety of processor core and specialized accelerators, to large-scale data-center systems, which feature racks of blades with general purpose processors, graphics-processor units (GPUs) and even accelerator boards based on FPGA technology. Furthermore, nowadays many embedded devices operate while being connected to one or more networks: e.g., modern video-game consoles rely on the Ethernet protocol [30], millions of TVs and set-top boxes are connected through DOCSIS networks [12], and most smartphones can access a variety of networks including 3G, 4G, LTE, and WLAN [17, 14, 32].

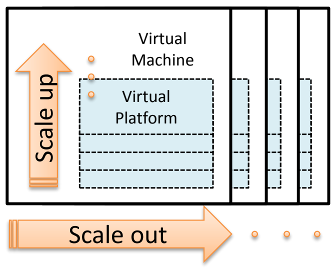


Fig. 1: The two orthogonal scalabilities of netShip.

As a consequence, a growing number of software applications involve computations that run concurrently on embedded devices and backend servers, which communicate through heterogeneous wireless and/or wired networks. For example, *mobile visual search* is a class of applications which leverages both the powerful computation capabilities of smart phones as well as their access to broadband wireless networks to connect to cloud-computing systems [11, 31].

We argue that the design and programming of these systems offer many new unique opportunities for the electronic design automation (EDA) community. For instance, system and sub-system architects need tools to model, simulate, and optimize the interaction of many heterogeneous devices; hardware designers need tools to characterize the applications, software and network stack that they must support; and software developers need early high-level modeling environments of the underlying hardware architecture, often much before all its components are finalized.

As a step in this direction, we present netShip, a networked virtual platform to develop simulatable models of large-scale heterogeneous systems and support the programming of embedded applications running on them. Users of netShip can model their target systems by combining multiple different virtual platforms with the help of an infrastructure that facilities their interconnection, synchronization, and management across different virtual machines.

Given a target system, netShip can be used to set up a simulation environment where each VP works as single-device simulator running a real software stack, e.g. the Linux operating system, with drivers and applications. Thus, it makes it possible to run real applications over the entire distributed system, without actually deploying the devices. This allows users both to jump start the functional verification process of the software and to drive the design optimization process of the hardware and the network.

While in certain areas the terms *virtual platform (VP)* and *virtual machine (VM)* are often used without a clear distinction, in this paper it is particularly important to distinguish them. A VP is a simulatable model of a system that includes processors and peripherals and uses binary translation to simulate the target binary code on top of a host instruction-set architecture (ISA). VPs enable system-level co-simulation of the hardware and the software parts of a given system before the actual hardware implementation is

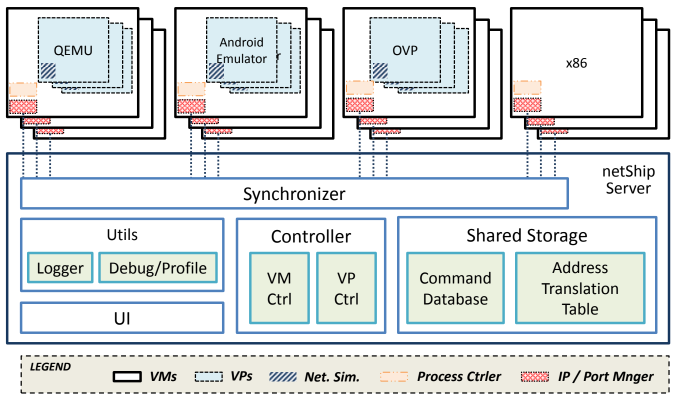


Fig. 2: The architecture of **NETSHIP.**

finalized. Instead, a VM is the management and provisioning of physical resources in order to create a virtualized environment. The resources are mostly provided by one or more server computers and the management is performed by a *hypervisor*. Examples of VPs include OVP, VSP, and QEMU, while KVM, VMware, and the instances enabled by the Xen hypervisor are examples of VMs. [1]

Thanks to its novel *VP-on-VM model*, the NETSHIP infrastructure simplifies the difficult process of modeling a system with multiple different VPs. In fact, the ability to support multiple VPs interconnected through a network makes NETSHIP free from the limitation of one specific VP while providing access to the superset of their features. For example, users who are interested in modeling an application running in part on certain ARM-based mobile phones and in part on MIPS-based servers can use NETSHIP to build a network of Android emulators [1] and OVP nodes.

The VP-on-VM model makes NETSHIP scalable both horizontally and vertically, as illustrated in Fig. 1. The users can scale the system out by adding more VM instances to the network (horizontal scalability) and scale the system up by assigning to each VM instance more CPU cores on which more VP instances can run (vertical scalability).

Another pivotal advantage the VP-on-VM model adds to NETSHIP is access to the features of VMs, i.e. pausing, resuming the VM instances, duplicating instanced preconfigured for specific VP types, or migrating them across physical machines.

Contributions. The main goal of this research work is to understand how to build and use a *Networked Virtual Platform* for the analysis of distributed heterogeneous embedded systems. To do so, we built NETSHIP as a prototype based on the VP-over-VM model with the main objectives of supporting heterogeneity and scalability. To the best of our knowledge, this is the first paper that presents this type of CAD tool. To evaluate NETSHIP we have completed a series of experiments including two complete case studies. The first case study shows how a networked virtual platform can be used to better utilize the computational resources that are available in the target system while guaranteeing certain performance metrics. The second case study shows how a networked virtual platform can be used to develop a software application running on a heterogeneous distributed system that consists of many personal mobile devices and multiple computer servers while, at the same time, obtaining an estimation of the resource utilization of the entire system.

2. NETWORKED VIRTUAL PLATFORMS

A heterogeneous distributed embedded system can consists of a network connecting a variety of different components. In our approach, we consider three main types of heterogeneity: first, we are interested in modeling systems that combine computing nodes based on different types of processor cores supporting different ISAs (*core-level heterogeneity*); second, nodes that are based on the same processor core may differ for the configuration of hardware accelerators, specialized coprocessors like GPUs, and other peripherals (*node-level heterogeneity*); third, the network itself can be heterogeneous, e.g. some nodes may communicate via a particular wireless standard, like GSM or Wi-Fi, while others may communicate through Ethernet (*network-level heterogeneity.*)

NETSHIP provides the infrastructure to connect multiple VPs in order to create a networked VP that can be used to model one particular system architecture having one or more of these heterogeneity levels. For example, Fig. 2 shows one particular instance of NETSHIP which is obtained by connecting multiple instances of the QEMU machine emulator [6], the Android mobile-device emulator [1], and the Open Virtual Platform (OVP) [3].

Each VP instance runs an operating system, e.g. Linux, with all the required device drivers for the available peripherals and accelerators. The application software is executed on top of the operating system. Each VP typically supports the modeling of a different subset of peripherals: e.g., OVP supports various methods to model the hardware accelerators of an SoC: users can write models in SystemC TLM 2.0 or take advantage of the BHM (Behavioral Hardware Modeling) and PPM (Peripheral Programming Model), which are C-compatible Application Programming Interfaces (APIs) that can be compiled using the OVP-supplied PSE toolchain[2].

In addition to the features supported by each particular VP, we equipped NETSHIP with all the necessary instrumentation to: (1) enable multiple instance executions; (2) configure port forwarding; and (3) measure the internal simulation time. Furthermore, any node in the network of VPs could potentially be a real platform, instead of being a virtual one: e.g. in Fig. 2, each of the x86 processors runs native binary code and still behaves as a node of the network.

One of the main novelty aspects of NETSHIP is the *VP-on-VM model* which is critical for the scalability of modeling and simulations. We designed NETSHIP so that multiple VP instances (e.g., 2 to 8) can be hosted by the same VM. By adding more VMs, the number of VPs in the system can be increased with a small performance penalty, as discussed in Section 3. Notice that the simple action of cloning a VM image that includes several VPs often represents a convenient way to scale out the model of the target system.

Next, we describe the main building blocks of NETSHIP.

Synchronizer. VPs vary in the degree of accuracy of the timing models for the CPU performance that they support. Some VPs do not have any timing model and simply execute the binary code as fast as possible. This is often desirable, particularly when a VP runs in isolation. In NETSHIP, however, we are running multiple VPs on the same VM and, therefore, we must prevent a VP from taking too much CPU resources and starving other VPs. QEMU provides a crude way to keep simulation time within a few seconds of realtime. OVP, instead, controls the execution speed so that the simulated time never surpasses the wall clock time. Multiple OVP instances, however, still show different time developments which require a synchronization method across the VPs in the network.

We equipped NETSHIP with a *synchronizer* module to support synchronization across the heterogeneous set of VPs in the networked platform, as shown in Fig. 2. The synchronizer is a single process that runs on just one particular VM and is designed in a way similar to the fixed-time step

[1]Recent efforts to run VMs on embedded cores [7, 19] remain within the VM definition as they do not adopt binary translation.

[2]PSE is Imperas Peripheral Simulation Engine [3].

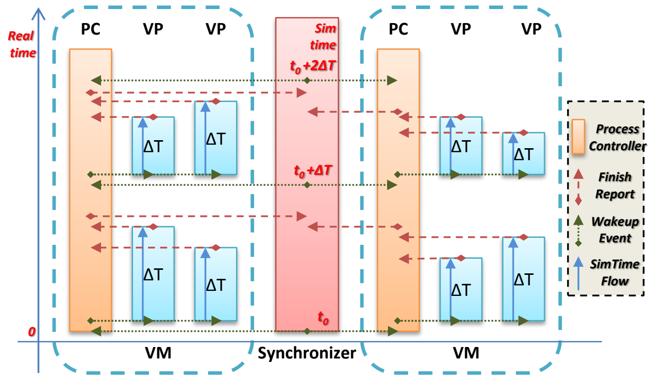


Fig. 3: Synchronization process example.

synchronization method presented in [8]: at each iteration, a central node increases the base timestamp and the client nodes stop after reaching the given timestamp. However, we considered two aspects in our synchronizer:

- we must synchronize VPs that might be scattered over several physically-separated machines;
- we must preserve the scalability provided by the VP-on-VM model.

NETSHIP targets large-scale systems which involve deployments across physically- separated machines where millisecond-level network packet travelling is actually required to synchronize. Hence, NETSHIP supports the modeling of applications that have running times ranging from a few seconds to multiple hours or days, rather than simulations at nanosecond-level.

To support synchronization over the VP-on-VM model, we designed a *Process Controller* (PC) that allows us to manage the VPs in a hierarchical manner. Each VM hosts one PC, which controls all the VPs on that VM. In particular, all messages sent by a VP to the synchronizer pass through the PC. The PC supports also running programs on a host machine: e.g. in the case of Fig. 2, the PCs manage the synchronization of the processes running on a x86 through the two POSIX signals `SIGSTOP` and `SIGCONT`, in the same way as the UNIX command `cpulimit` limits the CPU usage of a process.

Fig. 3 illustrates an example of the synchronization process with two VMs, each hosting two VP instances. The following steps happen at each given iteration i:

1. the synchronizer issues a future simulation time $t_i = t_{i-1} + \Delta T$ to the VPs and wakes them up;
2. the VPs run until they reach the appointed time t_i and report to their PC;
3. As soon as a PC receives reports from all the connected VPs, it reports to the synchronizer;
4. After the synchronizer has received the reports from all the PCs, it loops back to Step 1.

The users can configure the time step ΔT to adjust the trade-off between the accuracy and the simulation speed. We briefly discuss the complexity comparison of this hierarchical method in Appendix A.2.

Command Database. NETSHIP was designed to support the modeling of systems with a large scale of target networked VPs. In these cases, to manually manage many VP instances becomes a demanding effort involving many tasks, including: add/remove new VP instances to/from a system, start the execution of applications in every instance, and modify configuration files in the local storage of each instance. In order to simplify the management of the networked VP as a whole, we developed the *Command Database* that stores the script programs used by the different NETSHIP modules. For example, the network simulation module and IP/Port forwarding module load the corresponding scripts from the database and execute them. Table 8 contains a detailed list of the commands in the database.

VM and VP Management. Whereas the commands in the Command Database are dedicated to VP configuration, we developed specialized modules to manage the VPs and the VMs (for the latter we integrated tools provided by the VM vendor). These modules manage the disk images of the VMs and VPs, for creating, copying, and deleting their instances. Since many VPs are still in the early stages of development and are frequently updated by the vendors, the VP management module checks the availability of new updates for all the installed VPs.

Network Simulation. The VP models of NETSHIP are provided with their own models of the network interface card (NIC). These models, however, are purely behavioral and do not capture any network performance property, such as bandwidth or latency [8]. Consequently, we developed a *Network Simulation* module that enables the specification of bandwidth, latency, and error rates, thus supporting the modeling of network-level heterogeneity in any system modeled with NETSHIP. As shown in Fig. 2, a Network Simulation module resides in each particular VP and uses the traffic-shaping features based on the *tc* command, which manipulates the traffic control settings of the Linux kernel.

Address Translation Table. In NETSHIP there are two points where packet forwarding plays a critical role:

1. To allow incoming connections to the VPs through their emulated NIC model, most VPs provide a way to redirect a port of the host to a port of the VP, so that packets that arrive to that VM port are redirected to the corresponding VP port. We leverage this redirection mechanism so that the applications running on the VPs can open ports to receive packets from other VPs, even if those are located on a physically separated VM. [3] More details are described in the Appendix A.3.
2. Since certain applications required that each VP must be accessible through a unique IP address and generally there is only one physical IP address per VM, we must map each VP to a virtual IP address. Each VP must know such mapping for all other VPs in the system. Hence, we used the UNIX command `iptables` to create a table of assignments within the kernel of each VP. NETSHIP stores the translation information in the *Address Translation Table*, which is loaded through the network commands stored in the Command Database.

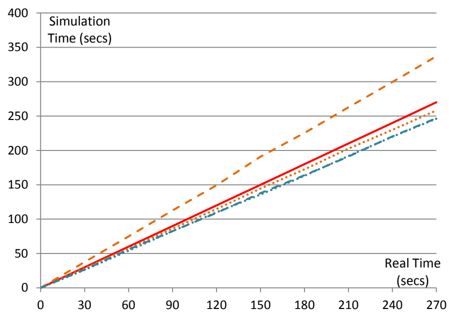
3. SCALABILITY EVALUATION

In this section, we experiment NETSHIP from the synchronization, scalability, performance, and network-fairness perspectives. Functional validations of NETSHIP will be covered in each case study, in Section 4 and 5.

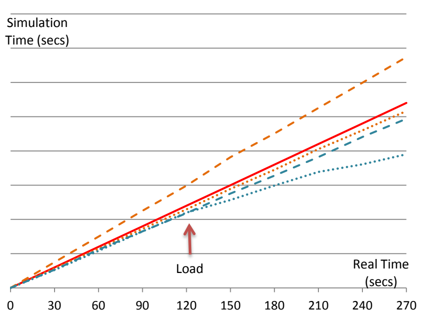
Simulated Time and Synchronization. Eight OVP instances and eight QEMU instances are running in this simulation setup. The three figures in Fig. 4 show the *simulated time* in each. The red solid line represents the time graph of an ideal VP, with $y = x$, where y is the *wall-clock time* and x is the simulated time. While there were multiple instances running together, in the figure we show only the fastest and slowest instances for each VP family, in order to summarize the range of variations within each VP family and to better compare the VP families.

Fig. 4(a) measures the simulated time of unloaded VPs. Each VP advances its simulated time linearly, but differently from each other. In particular, the range of simulated time among QEMU instances is wide: from 4% slower up to 25% faster than the wall-clock time. Instead, the OVP instances show almost the same simulation speed (0.3% variation), which is 8% slower than the slowest QEMU instance. This

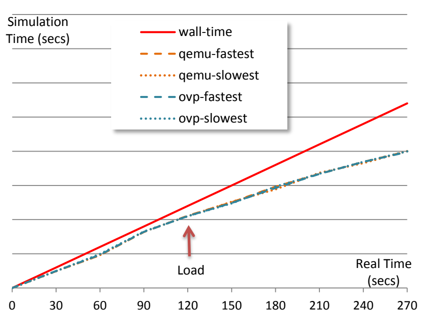
[3]While certain VPs provides a network bridge feature that allows more generic network functionalities, we use port redirection because it is commonly supported by every VP family.



(a) No load, No sync



(b) Load, No sync



(c) Load, Sync

Fig. 4: Simulation time measurements.

VP Type	Core Model	CPU use	Preferred #VPs
OVP	Accelerator	~ 24%	4
OVP	MIPS	~ 6%	16
QEMU	PowerPC	~ 12%	8
VMWare	x86	~ 5%	20

Table 1: Host CPU use of each VP.

reflects the fact that OVP has a better method to control the simulation speed.

Fig. 4(b) shows the case when a VP is subject to a heavy workload. In particular, at simulated time $x = 120s$ one OVP instance starts using a high-performance accelerator. From that point on, the OVP instance gets slower than every other instances, as shown by the deviation among the OVP lines in the figure. This is natural when the peripherals are modeled at a very high level of abstraction. In a fair host VM, all VPs are granted the same amount of CPU time to be executed. Simulating the use of a hardware accelerator on a VP typically requires the VP process on the VM to executes a non-negligible computation. In the other words, running the functional model of the accelerator uses the VM's CPU resources and requires a certain amount of wall-clock time. From the viewpoint of simulated time, however, this computation happens in a short period of time (due to the accelerator's timing model); therefore the given VP instance becomes slower than the others. The misalignment of the simulated time among VP instances is a concern when simulating distributed systems, because it might cause the simulated behaviors to be not representative of reality.

To address this problem, we implemented the synchronization mechanism explained in Section 2. Fig. 4(c) shows the behavior of all VP instances under the same conditions but with the synchronization mechanism turned on (with a synchronization cycle of 300ms). The simulated time of all VPs becomes the same as the slowest instance. The synchronization cycle can be decided by the users. Our experiments show that it should not be too small ($\geq$ 1ms) because: i) a synchronization that is much more frequent than the OS scheduling time slice[4] may disturb the timely execution of the VPs, and ii) the synchronization is an overhead and slows down the overall simulation.

Vertical Scalability. By *vertical scalability* we mean the behavior of the networked VP as more VPs are added to a single VM. As discussed above, although the synchronizer preserves the simultaneity of the simulation among VPs, it makes them all run at the speed of the slowest instance , i.e. even one slow VP instance is enough to degrade the simulation performance of the whole system. Therefore, an excessive number of VP instances on the same VM will likely cause a simulation slowdown.

Table 1 shows the amount of CPU of the host VM that is used by a VP instance. For example, when an OVP instance fully utilizes one accelerator, it takes up to 24% of the

[4]Linux O(1) scheduler dynamically determines the time slice, ranging from milliseconds to a few hundreds milliseconds.

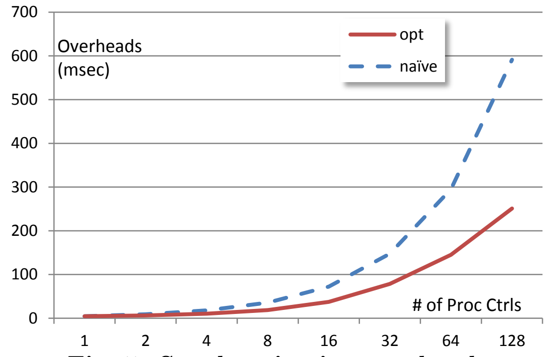


Fig. 5: Synchronization overheads.

host CPU resource in the hosting VM. This means that 4 is the optimal number of OVP instances, equipped with that accelerator, which can co-exist on the same VM without performance penalty. Likewise, the CPU of a QEMU PowerPC that is fully busy, i.e. a simulated 100% utilization, uses up to 12% of the host VM's CPU resources: hosting up to 8 QEMU PowerPC instances in the same VM is performance optimal.

Note that even if the number of VP instances goes over the optimal value, the synchronizer still preserves the simultaneous simulation of all nodes. However, balancing out the number of VP instances hosted across the VPs, or alternatively increasing the computational resources available to the VM, helps to increase the overall simulation performance.[5]

Synchronization Overheads and Horizontal Scalability. *Horizontal scalability* describes the behavior of the simulated VP as we scale the number of VMs. The synchronizer is the entity in the networked VP that communicates with all VMs in order to keep all VPs aligned. Fig. 5 shows the overhead increase as the number of VMs grows. We measured the overhead as the time elapsed from when the slowest VP instance reports to have terminated the execution step to the time the same instance starts the new one. We experimented with ten VP instances insisting on each PC. In the figure we compare a naïve implementation with an optimized implementation of the synchronizer. For both version the principle of the synchronization is the same; however, in the optimized version we used more advanced techniques to reduce the communication latency and overhead, such as multicasting wakeup, local reporting by atomic operations on a shared memory, and POSIX signals, as described in Appendix B.1. The overhead for 128 PCs is approximately $250ms$, which slows down the networked VP by about 25% if the simulation step is set to $1s$.

Although the optimized implementation significantly reduces the overhead, both slopes increase linearly with re-

[5]The CPU resources of the VM might not be the only bottleneck. For a more generic approach, an analysis of disks, network congestion, memory bandwidth, bus capacity, and cache interference are required. In our experiments, however, the constraint due to the VM's processing power was the most dominating factor that decides Vertical Scalability.

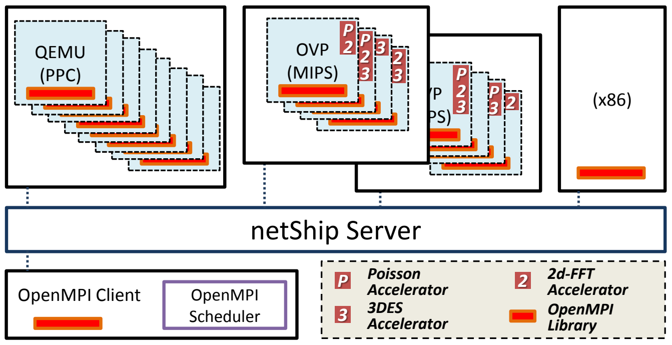


Fig. 6: The system architecture for Case Study I.

Algorithm	Operations per Hour		Speedup
	CPU	Accelerator	
Poisson	349	1183	3.39
2d-FFT	314	517	1.65
3DES	632	1339	2.12

Table 2: Case Study I: performance comparisons.

VP Type	Network Type	Bandwidth	Latency
OVP MIPS	DOCSIS 2.0	30.72Mbps	30ms
QEMU PowerPC	Evolved EDGE	1.00Mbps	80ms
Host x86	IEEE 802.11g	54.00Mbps	45ms

Table 3: Case Study I: configured bandwidth & latency.

spect to the number of PCs in the network (notice that the x-axis is logarithmic). This is because synchronization involves all PCs, each of which is located in separate machine, and all reports require a packet transmission across the network and linear-time computation to parse the reports.

In summary, the synchronization across VMs limits the horizontal scalability, in the sense that the simulation step after which all VPs are synchronized, must be (much) bigger than the time it takes to actually perform the synchronization, which strongly depends on the characteristics of the hosting VMs and how they are connected.

4. CASE STUDY I - MPI SCHEDULER

We modeled a distributed embedded system as a networked VP, which runs Open MPI (Message Passing Interface) applications. This system features all the three kinds of heterogeneities: it has three different models of CPUs, it has arbitrarily scattered accelerators over the VPs, and we also vary the types of network the devices are connected to. The goal of this case study is to use a networked VP for designing a static scheduler that optimizes the execution time by better distributing the work across the system in Fig. 6.

Open MPI is an open source MPI-2 implementation, which is a standardized and portable message passing system for various parallel computers [2]. In this case study, we used Open MPI to establish a computation and communication model over NETSHIP. Since the mainstream implementation of Open MPI does not support MIPS or ARM architectures (because it misses the implementation of atomic operation backends) we wrote and applied patches for Open MPI to run on these ISAs.

We simultaneously run three MPI applications over the distributed system: Poisson [22], 2d-FFT [29], and Triple DES [5]. Each application is a standalone executable program and is configured to process a small amount of data so that they act as embarrassingly parallel. Every application is designed to either use the hardware accelerator, whenever available on the VP, or run purely in software, otherwise. Accelerators are modeled to run basically the same algorithm as the applications. We modeled one iteration of the algorithm to take a few milliseconds.

According to our timing model for the accelerators and to the native timing model of the CPUs, accelerators show 1.65× ∼ 3.39× speedup over CPUs, as summarized in Table 2. Note that the speedup introduced by hardware acceleration with respect to pure software execution is not the main point here. Instead, the comparative analysis is a demonstration of the type of analysis that a designer can carry by using the networked VP paradigm. In fact, the speedup mentioned above is actually conservative with respect to the literature in order to keep the design exploration of our case study interesting [28, 27].

Based on such time model, Fig. 7 shows the performance profile for different applications on a few VPs: the OVP instances are always equipped with an accelerator, while the other VPs are not. We also consider models for the network, whose bandwidth and latency parameters are summarized in Table 3. Note that, since NETSHIP allows designers to use time models to better simulate the characteristics of the network, those models are inputs to the networked VP and their derivation goes beyond the scope of this paper.

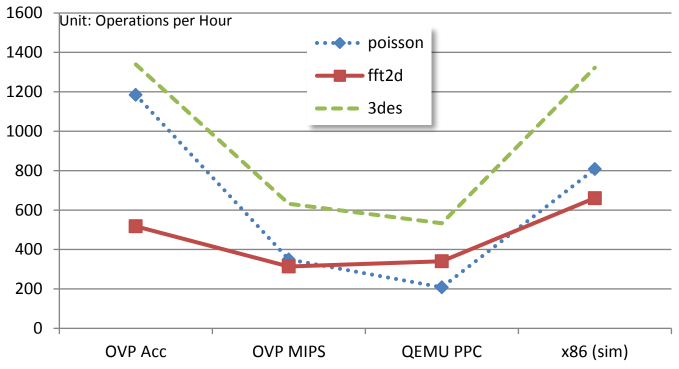


Fig. 7: Case Study I: performance of different cores.

In order to improve the application performance by taking advantage of the known properties of the system, i.e. performance profile of the nodes and network characteristics, we designed an *OpenMPI Scheduler* and we used the networked VP to evaluate its effectiveness. [6] As shown in Table 4, the scheduler delivers a speedup ranging from 1.3× to over 4×, depending on the user request. Such achievement is very encouraging since it is obtained without using any additional resource, but only by re-assigning tasks to the nodes that are better equipped for each of them. Note that the design, the verification, and also an initial assessment on the effectiveness of the scheduler have been carried out on the networked VP, without having to deploy the real system.

5. CASE STUDY II - CROWD ESTIMATION

Crowd estimation, or crowd counting, is the problem of predicting how many people are passing by or are already in a given area [21]. A number of researches have focused on the crowd estimation based on image processing of pictures [9, 20]. The crowd estimation application we developed in this section is based on user-taken pictures, from mobile phones, targeting relatively wide areas, e.g. a city.

We built a networked VP (Fig. 8) that is representative of the typical distributed platform required to host this kind of application. The networked VP features *Android Emulator*s to model the phones and a cluster of *MIPS-based servers* based on the multiple OVP instances (on the right-hand side of the figure). The Android Emulators emulate mobile phones that take pictures through the integrated camera and upload them to the cloud. The pictures are stored

[6]Details on the scheduler design are available in Section C of the Appendix.

# of operations			Time in ms		
Poisson	2d FFT	Triple DES	without Sched	with Sched	Speed Up
60	30	800	199,642	48,924	4.08
60	30	1800	344,422	117,380	2.93
50	20	40	102,293	45,210	2.26
60	30	150	103,700	46,693	2.22
250	80	140	198,927	113,383	1.75
20	100	10	161,462	122,527	1.32

Table 4: Case Study I: scheduler performance.

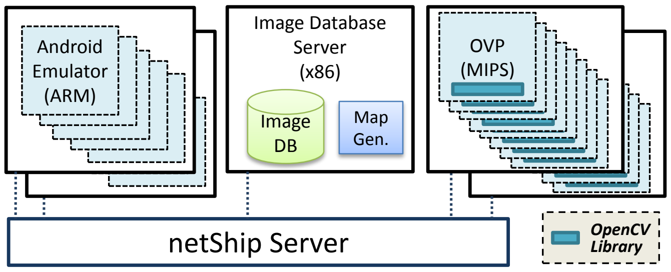


Fig. 8: The system architecture for Case Study II.

on an *Image Database Server* (IDS), to which both phones and servers have access. The servers emulate the cloud, and run image processing algorithms on the pictures. Specifically, we developed a *Human Recognition* application based on OpenCV [4] to count the people in each picture and store the result on the IDS. Then, a *Map Generator* process running on the IDS reads the people counting from the IDS and plots it on a map.

Given the application requirements, we used the networked VP to gain insights on the amount of resources required to process pictures in real-time. Note that our main concern is the opportunity to build and study the networked VP, and to use it to analyze the properties of the application that runs on it. In other words, we used this application primarily as a case study to test the capabilities of NETSHIP, while the optimization of the quality of the crowd estimation was only a secondary concern.

Android Emulator Scalability. We used several Android Emulators to model millions of mobile phones that sporadically take pictures (instead of using millions of emulators). To validate whether the emulators realistically reflect the actual devices' behavior with respect to network utilization, we performed multiple tests after making the following practical assumptions:

1. There are 3,000,000 mobile phone users in Manhattan and 2% of them upload 2 pictures a day.
2. The uploading of the pictures is evenly spread over the daytime (09:00~ 18:00).
3. The average image file size is 74KB, as the image size we have in the DB.

Given the assumptions above, we summarize in Table 5 the number of pictures an emulator must upload in an hour and the actually measured incoming traffic of the DB server, with respect to the number of available emulators in the networked VP and accordingly configured the number of pictures uploaded by each emulator per hour. For example, if the networked VP has only one Android emulator (first row), we can achieve the desired load for the cluster when this emulator uploads $3,000,000 * 0.02 * 2/9 \approx 13333$ pictures per hour. Since one single emulator fails to upload 13333 pictures per hour, because of insufficient emulator performance, we must increase the number of emulators to at least 2. The measured incoming traffic is rather consistent independently of how many emulators we use to split the job. This implies that we can deploy less emulators than the number of nodes

# of Emulator in the model	# of Pic Upload / Hour	Incoming Traffic (KB/s) Max	Min	Avg.
1	13333	N/A	N/A	N/A
2	6666	380.4	372.5	379.3
4	3333	384.1	376.4	381.2
8	1666	377.8	361.8	374.2
16	833	389.0	367.5	381.5

Table 5: Case Study II: impact of varying number of Android-emulator instances.

Image Size (KB)	8	32	128	512	74(Avg)
Process Time (s)	3.48	13.42	49.7	247.1	31.5
Throughput (KB/s)	2.30	2.38	2.57	2.07	2.34

Table 6: Case Study II: image processing (human recognition) performance.

we would have in reality, i.e., 4 vs. 3 million, as long as those emulators generate more traffic than they would in reality, i.e. 3333/hour vs. 2/day, after verifying that they also simulate fast enough to sustain the traffic generation. We leave the modeling of more complicated traffic patterns than *Assumption 1*, e.g. bursty traffic, as future work.

Bottleneck Analysis. The data in Table 6 show the average time required by one MIPS server to run the Human Recognition application on a given picture. Based on this data and on the characterization of the traffic load, the application designer can attain a number of meaningful design considerations.

1. The designer can measure the number of required MIPS servers that support the required volume of image processing, given the input and output data rates. For example, when images are fed to the DB at $380KB/s$, then, based on the throughput for the average image size in Table 6, the cluster must have more than $380KB/2.34KB/s = 162.4$ MIPS servers to guarantee real-time performance. Note that $2.34KB/s$ is the throughput of the average image size in Table 6.
2. On the other side, if the number of available servers cannot be changed, the designer can reason on the appropriate image size. If we assume to have 80 MIPS servers in the cluster, then they can process only up to $80 \times 2.34KB/s = 187.2KB/s$. In that case, the average image size must be less than $74KB \times (187.2/380) = 36.5KB$ for the application to work in real-time.
3. The network traffic through the DB server includes picture uploading from mobile phones, picture downloading by the MIPS clusters, updating and reading of geolocation information and people count. Based on an analysis of the network traffic and how it scales as the system grows, the designer can evaluate the best database architecture, e.g. distributed rather than centralized.

6. RELATED WORK

A number of studies have previously focused on helping system architects to better design distributed embedded systems by providing ways to optimize the process scheduling and the communication protocols [15, 23], tools to ease design space explorations [26, 16, 13], estimation models [34], and network behavior simulations [10], or methodologies [18, 13]. Nonetheless, these tools or systems only generate quantitative guidelines that must be then applied to the physical devices, thereby precluding their usability without already having the physical devices in place and the application deployed on them. There are also contributions obtained through the use of VPs [8, 33]; however, none of these works consider the three levels of heterogeneity that characterize more and more distributed embedded systems (Section 2).

Synchronization between the VP instances, one of the key features of our networked VP, has been inspired by [25, 24]. However, they do not consider node-level or network-level heterogeneity.

7. CONCLUSIONS

We have designed and implemented netShip, a framework for building networked VPs that model heterogeneous distributed embedded systems. Networked VPs can be utilized for various purposes, including: i) simulation of distributed applications, ii) systems, power, and performance analysis, and iii) costs modeling and analysis of embedded networks' characteristics.

We also designed hardware accelerators for specific algorithms. We analyzed that accelerators might require more resources of the CPUs that host the simulation. We quantified how this phenomenon partially limits the scalability of the entire networked VP, and provided guidelines on how to distribute the VPs in order to counter balance this loss of simulation performance.

Finally, we used netShip to develop two networked VPs. We used one VP to design a scheduler based on MPI and to verify through simulation how the scheduler is able to optimize the execution of many MPI jobs over a network of heterogeneous machines, by simply distributing the jobs among the available machines on the basis of their performance-per-application profile. We used the other VP to design and validate an application distributed among portable devices and a cloud of servers, and also to derive potential insight about the number of servers and the image size that guarantee the entire application to run in real-time.

Acknowledgements

This project is partially supported by the National Science Foundation under Awards #644202 and #1147406 and by an ONR Young Investigator Award. We gratefully acknowledge Yosinori Watanabe for useful discussions.

8. REFERENCES

[1] Android (`developer.android.com`).
[2] Open MPI (`www.open-mpi.org`).
[3] Open Virtual Platforms (www.ovpworld.org).
[4] OpenCV (`opencv.org`).
[5] W. C. Barker and E. B. Barker. SP 800-67 Rev. 1. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. Technical report, Jan. 2012.
[6] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, pages 41–46, Feb. 2005.
[7] C. Dall and J. Nieh. KVM for ARM. In *Proc. of the Linux Symp.*, pages 45–56, July 2010.
[8] M. D.Angelo et al. A simulator based on QEMU and SystemC for robustness testing of a networked Linux-based fire detection and alarm system. In *Proc. of the Conf. on ERTS^2*, pages 1–9, Feb. 2012.
[9] T. Fei, L. SunDong, and G. Sen. A novel method of crowd estimation in public locations. In *Int. Conf. on FBIE*, pages 339–342, Dec. 2009.
[10] E. Giordano et al. MoViT: the mobile network virtualized testbed. In *Proc. of the Int Workshop on VANET*, pages 3–12, June 2012.
[11] B. Girod et al. Mobile visual search. *IEEE Signal Processing Magazine*, 28(4):61–76, July 2011.
[12] S. Howard and J. Martin. DOCSIS performance evaluation: piggybacking versus concatenation. In *Proc. of Southeast Regional Conf.*, volume 2, pages 43–48, Mar. 2005.
[13] Z.-M. Hsu, J.-C. Yeh, and I.-Y. Chuang. An accurate system architecture refinement methodology with mixed abstraction-level virtual platform. In *Proc. of DATE*, pages 568–573, Mar. 2010.
[14] M. Ismail. WiMAX: a competing or complementary technology to 3G? In *Proc. of the Conf. on Integrated Circuits and Syst. Design*, pages 3–3, Sept. 2007.
[15] V. Izosimov et al. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Proc. of the conf. on DATE*, pages 864–869, Mar. 2005.
[16] D.-I. Kang et al. A software synthesis tool for distributed embedded system design. In *Proc. of the Workshop on Languages, Compilers, and Tools for Embed. Syst.*, pages 87–95, May 1999.
[17] D. Koutsonikolas and Y. C. Hu. On the feasibility of bandwidth estimation in wireless access networks. *Wireless Net.*, 17(6):1561–1580, Aug. 2011.
[18] J. Kruse et al. Introducing flexible quantity contracts into distributed SoC and embedded system design processes. In *Proc. of DATE*, pages 938–943, Mar. 2005.
[19] S.-M. Lee et al. Fine-grained I/O access control of the mobile devices based on the xen architecture. In *Proc. of the Int. Conf. on Mobile Comp. and Net.*, pages 273–284, Sept. 2009.
[20] W. Li et al. Crowd density estimation: An improved approach. In *Int. Conf. on Signal Processing*, pages 1213–1216, Oct. 2010.
[21] A. Marana et al. Estimating crowd density with Minkowski fractal dimension. In *Proc. of ICASSP*, volume 6, pages 3521–3524, Mar. 1999.
[22] N. Ng, N. Yoshida, and K. Honda. Safe parallel programming with message optimisation. In *Proc. of Int. Conf. on Objects, Models, Components, Patterns*, volume 7304, pages 202–218, May 2012.
[23] T. Pop, P. Eles, and Z. Peng. Design optimization of mixed time/event-triggered distributed embedded systems. In *Proc. of CODE+ISSS*, pages 83–89, Sept. 2003.
[24] D. Quaglia et al. Timing aspects in QEMU/SystemC synchronization. *Proc. of the Int. QEMU Users' Forum*, pages 11–14, Mar. 2011.
[25] H. Raj et al. Enabling semantic communications for virtual machines via iConnect. In *Proc. of the Int. Workshop on VTDC*, pages 1:1–1:8, Nov. 2007.
[26] D. E. Setliff, J. K. Strosnider, and J. A. Madriz. Towards a design assistant for distributed embedded systems. In *Proc. of the Int. Conf. on ASE*, pages 311–312, Nov. 1997.
[27] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. *SIGARCH Comp. Arch. News*, 19(1):106–113, Mar. 1991.
[28] T. Shimokawabe et al. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *Proc. of the Int. Conf. for High Perf. Comp., Net., Storage and Anal.*, pages 1–11, Nov. 2010.
[29] R. C. Singleton. On computing the fast Fourier transform. *Comm. of the ACM*, 10(10):647–654, Oct. 1967.
[30] D. Taylor. Need for speed: PS3 Linux! *Linux Journal*, (156):5–6, Apr. 2007.
[31] S. S. Tsai, D. Chen, J. P. Singh, and B. Girod. Rate-efficient, real-time cd cover recognition on a camera-phone. In *Proc. of the 16th ACM Intl. Conf. on Multimedia*, pages 1023–1024, Oct. 2008.
[32] C. H. K. van Berkel. Multi-core for mobile phones. In *Proc. of DATE*, pages 1260–1265, Mar. 2009.
[33] C.-C. Wang et al. NetVP: A system-level network virtual platform for network accelerator development. In *IEEE Int. Symp. on Circuits and Systems*, pages 249–252, May 2012.
[34] Y. Xiangzhan et al. Research on performance estimation model of distributed network simulation based on PDNS conservative synchronization mechanism in complex environment. In *Proc. of the ICCIT*, pages 2576–2580, Dec. 2010.

APPENDIX

A. ARCHITECTURE

A.1 Scalability and Detailed Configurations

In general, Horizontal Scalability is the ability to have more VP instances running in netShip by adding more VM instances. Vertical Scalability is the ability to have more VP instances running in netShip by adding more CPU cores to a VM. For example, the preferred number of OVP instances with accelerators that can run in one VM (with one CPU core) is four, as shown in Table 1. If we add one more CPU core to the VM, we can run up to eight OVP instances with accelerators in that VM.

The configuration of Case Study I, shown in Fig. 6, includes: one VM that runs eight QEMUs, two VMs that run four OVPs each, and one VM that supports x86. This is an optimal configuration for the purpose of this case study. However, we also tested Horizontal Scalability (e.g. by adding one VM that runs eight other QEMUs and another VM that runs four other OVPs) and Vertical Scalability (e.g. by adding one CPU core to the VM running eight QEMUs so that it can sustain up to 16 QEMUs.)

For Case Study II we varied the number of Android Emulator in "Android Emulator Scalability" and the number of OVP instances in "Bottleneck Analysis " in Section 5. When we run 1 ~ 4 Android Emulator, we used one VM, for 8 two VMs (5 + 3), for 16 four VMs (5 + 5 + 5 + 1, because the preferred VP number of Android Emulator is 5). For VMs running OVP instances, we have used two CPU cores for each VM, hosting eight OVPs on each instance, for a total of four VMs for 32 OVP instances.

A.2 Synchronization Complexity Comparison

In the synchronization algorithm in [8], if the number of VP is $|VP|$ the synchronization process should receive and count $|VP|$ reports to make sure that all the VPs have reached to the appointed simulation time. This results in a $\Theta(|VP|)$ algorithm complexity in *Synchronizer*, whereas in netShip it is $\Theta(\sqrt{|VP|})$ because *Synchronizer* manages $\sqrt{|VP|}$ PCs, each of which controls $\sqrt{|VP|}$ VPs.[7]

A.3 Port Forwarding

Port forwarding is the technique of redirecting the traffic incoming on one network port of the OS running on the host VM towards a specific port of the OS running on the hosted VP. For example, when a packet arrives to Port 10020 of the VM's OS, the VP to which Port 10020 is assigned intercepts the packet and forwards it to Port 22 of the VP's OS. Hence, when users connects through SSH to the host's IP and Port 10020 they are forwarded to Port 22 of the VP. This is configured in the behavioral model of the VP and performed through the NIC model.

Unlike SSH, some libraries require a random port to be accessed by clients; for instance Open MPI communicates through random ports ranging from 1025 to 65535 [A4]. However, most libraries also provide a way to change or reduce the required port range as shown in Table A.3. We reduced the range and mapped it to the same port range on the virtual addresses, 200.0.0.x. One of these addresses is allocated to each of VP instances using *iptables* through the Port Management module in Fig. 2.

[7]It may be enough for *Synchronizer* only to count the number of reports from PC to know that every VP instance is ready and advance the simulation time. However, this method is unreliable in the sense that there is no way for *Synchronizer* to tolerate a PC malfunctioning. If a hash table, for example, is used to map a PC's IP to the data structure for checking that the PC is reporting more than once in a cycle, the average complexity of the algorithm in [8] is $O(|VP| + |VP|^2/k)$ and for our algorithm it is $O(\sqrt{|VP|} + \frac{|VP|}{k})$, where k is the number of buckets in the hash table and searching n times in a hash table takes $n * O(1 + n/k)$.

Library	Option	Default Value
SSH (fixed)	Port	22
Hadoop (fixed)	dfs.http.address	50070
	dfs.datanode.http.address	50075
	mapred.job.tracker.http.address	50030
	mapred.task.tracker.http.addres	50060
Open MPI (random)	oob_tcp_port_min_v4	0
	oob_tcp_port_range_v4	65535
	btl_tcp_port_min_v4	1025
	btl_tcp_port_range_v4	65525

Table 7: Example of library port uses.

A.4 Command Database

Name	Behavior
vp_ctrl_pwr	turns the VP on/off
net_set_bw	sets the VP's network bandwidth to simulate
net_set_delay	sets the VP's network delay to simulate
net_set_error	sets the VP's network error rate to simulate
net_load_rt	loads the address/port settings to use
cmd_execute	executes a command in all the VPs
acc_gen	loads driver modules and creates a device node for the specified accelerator
report_local	reports the local time in the VP
report_cpu	reports the cpu time in the VP

Table 8: List of commands in the command database.

B. EXPERIMENTS

B.1 Optimizing Synchronization

Multicasting-based Wakeup. In order to reduce the serial latency of the wakeup packets delivered from the synchronizer to the PCs, we used multicast UDP.

Atomic Operations in Shared Memory for In-Machine Reporting. The PC must check that VPs correctly report the end of the current simulation cycle. This is done by having each VP increase a shared counter through an atomic operation. This is possible because all VPs are on the same machine.

Disabling Nagle's Algorithm. Unlike the waking-up message of the synchronizer to the PCs (1-to-N), multicast UDP cannot be used to carry reports from the PCs to the synchronizer (N-to-1). In the Linux kernel, TCP sockets typically use by default an optimization technique, Nagle's algorithm, which combines a number of small outgoing packets and sends them all in one single message [A5]. This method, however, increases the latencies of these small packets (up to 30ms in our experiments), which is a critical issue in our synchronization design, since latency is way more important than throughput. We then disabled the Nagle's algorithm by turning on the socket option TCP_NODELAY for each TCP socket.

Using POSIX Signals to Sleep and Wake up. In order to stop and wake-up a VP instance our PC uses two signals: *SIGSTOP* and *SIGCONT*. The use of standard Linux signals provides several advantages. First, the PC can be easily implemented in a separate user space program, without the knowledge of the internals of the VPs. Second, once implemented, the PC is portable across the VPs, requiring no modifications. Third, the PC can stop all threads in the process, while sleeping works only for the thread of the current context. Most importantly, this also enables a synchronized execution with processes that run natively on a host VM, e.g. x86 server, outside of any VP.[8]

B.2 Network Fairness Depending on Deployment

[8]In netShip x86 binaries are executed on a VM, not a VP. Through the stop and continue signals PC synchronizes the process without modifying the binary executables.

Target	PNI[9]	RTT (ms)
self VP (the loopback interface)	no	0.15
local VM (a VM where the testing VP runs)	no	0.17
local VP (another VP on the local VM)	no	0.19
remote VM (a VM, except the local VM)	yes	0.17
remote VP (a VP on the remote VM)	yes	0.19

Table 9: Ping test from a VP.

Based on the measurement of the latency of packet responses, through tools such as *ping* or *traceroute*, it is possible to determine whether two VM instances are deployed on the same physical machine or not [A6]. Likewise, VP instances may experience variations in the network latency, depending on how they are deployed.

However, our experimental results in Table 9 show that, given a VP instance, the difference in latency to reach a *local VP* on the same VM, versus a *remote VP* on another VM, is $\leq 0.05ms$. In particular, this value is small enough to be effectively hidden by the latencies set by the designer to model the network-level heterogeneity configuration, as shown in Table 3.

C. CASE STUDY I

Scheduler Design. We designed the scheduler to run on a client machine and to follow these steps:

1. It receives the *user request*, which includes the number of times each MPI application must run, e.g. 300 Poissons, 200 2d-FFTs, and 500 3DESs.
2. It loads the *performance profile* of each VP in executing the given applications, e.g. VP_0 takes 3.182s to execute Poisson, while VP_1 takes 10.427s , and so on, as shown in Fig. 7.
3. It derives the objective function to be minimized.
4. It converts the constraints (user request and performance profile) and the objective function into a matrix, and solves it by running a linear programming algorithm.
5. It distributes the workload according to the solution.

Examples of the constraints and the objective function to 1) minimize the total execution time and 2) minimize the power dissipation are illustrated in Appendix C.1.

C.1 Linear Programming Examples

For the sake of simplicity, the following two examples assume that there are two devices, d_1 and d_2, and we have two applications, x_1 and x_2. In both examples, variables T_{ij} denotes the amount of time that device d_i spent on an application x_j. For instance, a variable T_{12} is the time period that device d_1 spent running application x_2. After executing a linear programming algorithm, the solution includes the best (the minimun or the maximun) possible value of the objective function along with the values of the variable used for the best value of the objective function.

Minimizing the Execution Time. Let's assume we have the following profile data:

1. application x_1 runs on device d_1 110 times per unit time.
2. application x_2 runs on device d_1 250 times per unit time.
3. application x_1 runs on device d_2 150 times per unit time.
4. application x_2 runs on device d_2 100 times per unit time.

The user's request may be like the following:

1. Execute application x_1 400 times and application x_2 320 times.

Then we have two inequations from them:

1. $110T_{11} + 150T_{21} >= 400$
2. $250T_{12} + 100T_{22} >= 320$

To bring the total execution time in the calculation, we introduce a new variable t.

1. $T_{11} + T_{12} <= t$
2. $T_{21} + T_{22} <= t$

Finally, the objective function p to be minimized will be equal to t. The resulted matrix is given in Table 10.

$T11$	T_{12}	T_{21}	T_{22}	t	
110	0	150	0	0	400
0	250	0	100	0	320
-1	-1	0	0	1	0
0	0	-1	-1	1	0
0	0	0	0	1	0

Table 10: Linear programming matrix for minimizing execution time.

The optimal solution for this example is $p = 52/25$; $T_{11} = 4/5$, $T_{22} = 32/25$, $T_{21} = 52/25$, $T_{22} = 0$, $t = 52/25$. This means that the devices can finish the user request in 52/25 time units when the system follows this solution. The solutions for each variable, as the definition, stand for the execution time, e.g. T_{11} requires device d_1 to run application x_1 for 4/5 unit time or 88 times.

Minimizing the Power Dissipation. In this example, we assume the same user request and the profile data used in the execution time minimization example. In addition to these conditions, the power dissipation profile data is required:

1. device d_1 dissipates 30W per unit time when executing application x_1.
2. device d_1 dissipates 50W per unit time when executing application x_2.
3. device d_2 dissipates 20W per unit time when executing application x_1.
4. device d_2 dissipates 70W per unit time when executing application x_2.

Then the objective function derived is $p = 30T_{11} + 50T_{12} + 20T_{21} + 70T_{22}$. The resulted matrix to minimize this objective function subject to the constraints is given in the Table 11.

T_{11}	T_{12}	T_{21}	T_{22}	
110	0	150	0	400
0	250	0	100	320
30	50	20	70	0

Table 11: Linear programming matrix for minimizing power dissipation.

The execution of a linear programming algorithm for this matrix gives the solution $p = 352/3$; $T_{11} = 0$, $T_{12} = 32/25$, $T_{21} = 8/3$, $T_{22} = 0$. This means that the user requests can be executed with consuming 352/3 power units.

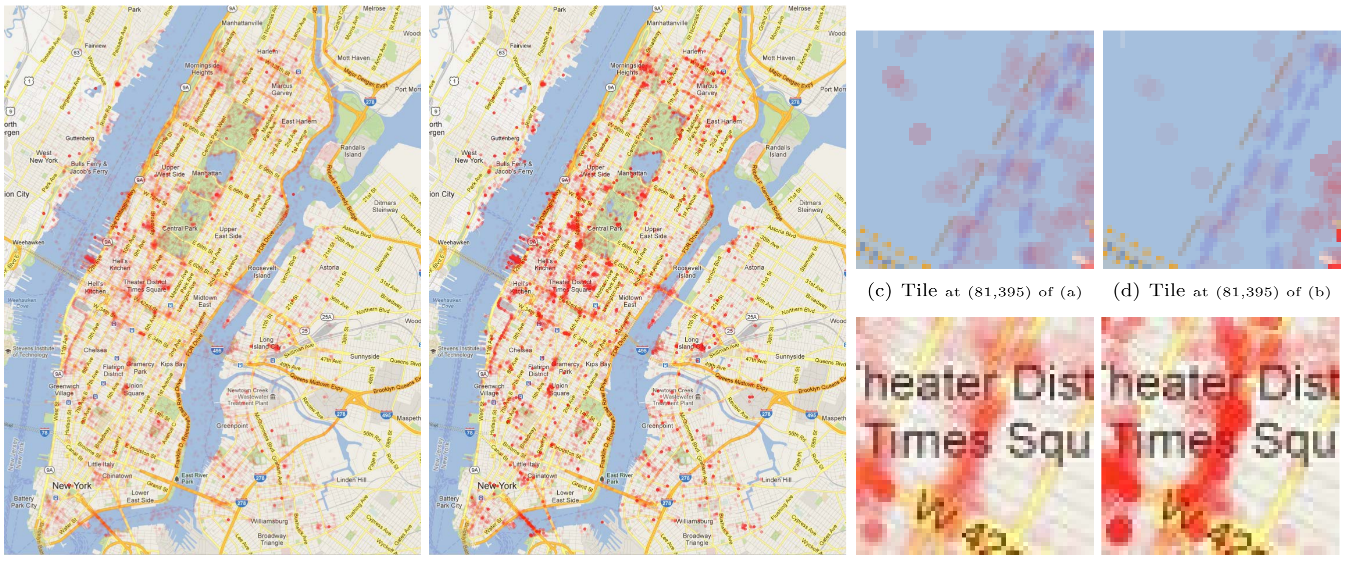
D. CASE STUDY II

D.1 Application Design

The application iterates the following work flow:

1. The mobile phone users take pictures and upload then to the Image DB along with their geolocation.
2. The cluster of MIPS servers fetches one image at the time from the DB and counts the people in it, by means of a human recognition algorithm.
3. The number of people in each image is stored back into the DB.
4. The Map Generator creates a plotted image as the result.

[9]Physical Network Involved.



(a) Picture Count (b) Crowd Estimates (c) Tile at (81,395) of (a) (d) Tile at (81,395) of (b) (e) Tile at (210,460) of (a) (f) Tile at (210,460) of (b)

Fig. 9: Case Study II: visualization of (a) picture count and (b) estimated crowd based on the pictures.

Each iteration is done in parallel, in the sense that the multiple Android Emulators upload images and the MIPS servers process the images concurrently.

The application consists of the following modules.

Android Camera App. One instance of the Android Camera App runs on each Android Emulator. To simulate the smart phones that users use to take pictures, we took images publicly available on the Picasa and Flickr's image databases [A1,A2], and we distributed them across the local storage of the Android Emulators, before starting the simulation. For the experiment, we considered 55,831 images with the geolocation information of Manhattan[10], assuming the users are taking pictures in this area. We modeled the act of a phone user taking a picture with the App loading a picture from the local storage.

Image Database. Every time the App takes a picture, it immediately uploads it, together with its metadata, i.e. latitude and longitude, to the Image DB. Also, the MIPS cluster fetches images and metadata from this database to process them, and stores back the results.

Human Recognition. The human recognition program is based on OpenCV. It is stored in the NETSHIP Server's storage, and runs on the MIPS cluster. To detect human bodies in a given picture, we used a head-and-shoulder detecting Haar model [A7] and an upper-and-lower-body detecting Haar model [A3]. It is, however, difficult to grasp human bodies from multiple directions, in particular from a side view [A8].

Map Generator. The Map Generator program reads the people counting from the Image DB and plots it on the map[11] with a resolution 717×944, translating latitude and longitude to the pixel position.

D.2 Experiments

Although the quality of the developed application's result is not the primary concern of this work, we present the resulted maps from two possible alternative variations of the crowd estimation application: one based on counting only *the number of pictures* taken at a particular location, and the other based on counting *the number of people* showing in those pictures. The results of Fig. 9 are interesting but they can be substantially improved by using NETSHIP to analyze various possible optimizations of the application.

Fig. 9(a) shows how many pictures are taken by users and Fig. 9(b) shows the estimated crowds based on such pictures. One red circle on the map corresponds to an area of approximately $2500m^2$ or $2990yd^2$. The density of the crowds is presented with the opacity, where the transparent area indicates no people and an opaque circle indicates more than 80 people in that area.

Fig. 9(c) is a tile taken from Fig. 9(a) at the pixel position <81,395> and Fig. 9(d) is a tile taken from Fig. 9(b) at the same position. Likewise, Fig. 9(e) is a tile taken from Fig. 9(a) at the pixel position <210,460> and Fig. 9(f) is from the same pixel position of Fig. 9(b).

Both Fig. 9(a) and Fig. 9(b) give the idea about which areas are more crowded than others, based on the opacity of the red circles on the map. However, the comparison of these two figures shows that our crowd estimation algorithm, based on human recognition, gives a more accurate outcome than simply counting the total number of taken pictures. For example, as shown in the comparison of the pair of Fig. 9(c) and Fig. 9(d), the estimated crowds on the river was decreased by the human recognition algorithm because the pictures taken over the river are mostly Manhattan skyline photos taken on a boat, a helicopter, or an airplane. On the other hand, in the case of Fig. 9(e) and Fig. 9(f), the actual crowds on the ground might be greater than the number of pictures taken on the same spot.

[10]For the geolocation of Manhattan we used longitude -74.015 ~ -73.928 and latitude 40.700 ~ 40.816.

[11]The map is extracted from the Google Maps service.

E. REFERENCES

[A1] The Flickr API (www.flickr.com/services/api).

[A2] The Picasa Web Albums Data API (developers.google.com/picasa-web).

[A3] H. Kruppa, M. Castrillon-Santana, and B. Schiele. Fast and robust face finding via local context. In *Proc. of the Int. Workshop on VS-PETS*, pages 157-164, Oct. 2003.

[A4] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: a flexible high performance MPI. In *Proc. of the Int. Conf. on PPAM*, pages 228-239, Feb. 2006.

[A5] J. Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Comp. Comm. Rev.*, 14(4):11-17, Oct. 1984.

[A6] T. Ristenpart et al. Hey, you, get off my cloud: exploring information leakage in third-party compute clouds. In *Proc. of the ACM Conf. on Comp. and Comm. Sec.*, pages 199-212, Nov. 2009.

[A7] M. C. Santana et al. Face and facial feature detection evaluation. In *Proc. of VISAPP*, pages 167-172, Apr. 2008.

[A8] P. Viola, M. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. *In Proc. of ICCV*, volume 2, pages 734-741, Oct. 2003.