MobiTest : An evaluation infrastructure for mobile distributed applications

by

Anirudh Sivaraman Kaushalram
Bachelor of Technology in Computer Science and Engineering
Indian Institute of Technology, Madras, 2010

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2012

Author .....
Department of Electrical Engineering and Computer Science
August 29, 2012

Certified by .....
Li-Shiuan Peh
Associate Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

MobiTest : An evaluation infrastructure for mobile distributed applications

by

Anirudh Sivaraman Kaushalram

Bachelor of Technology in Computer Science and Engineering

Indian Institute of Technology, Madras, 2010

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Sophisticated applications that run on mobile devices have become commonplace. Within the wide realm of mobile software applications there exists a significant number that make use of networking in some form. Unfortunately, such distributed mobile applications are inherently difficult to evaluate. Conventional evaluations of such distributed applications are limited to small, real-world deployments consisting of, perhaps, a handful of phones. Such tests often do not have the requisite number of users to produce the desired performance. Also, these experiments do not scale and are not repeatable. To address all these issues, we sought to evaluate distributed applications in a virtual environment. Besides being cheaper, such evaluations are reproducible and scale significantly better. This thesis documents our efforts in working towards this goal. We discuss the designs that we iterated through, along with the problems we faced in each of them. We hope these problems will inform future designs that can solve the challenges that we weren't able to solve efficiently.

Thesis Supervisor: Li-Shiuan Peh
Title: Associate Professor

Acknowledgments

Several people have been of great help to me over the last two years that went into my Master's thesis. The list begins with my advisor, Li-Shiuan Peh, who took me in as a first year graduate student here. Li-Shiuan gave me immense freedom in defining my thesis topic and at the same time ensured that I was making progress. Her unbounded optimism and constantly cheerful demeanor are qualities that I hope to emulate. I always came out of a meeting with her feeling better about my research.

My academic advisor, Piotr Indyk, has been a great sounding board for issues relating to research and grad school. He was always willing to take time out to meet me whenever I was confused.

Niket Agarwal was the first person I spoke to about research once I joined MIT. He was always ready to discuss any aspect of his research despite the fact that he was close to graduating by the time I joined MIT.

Jason Gao has been a great collaborator and more importantly, a great labmate. His penchant for getting things done in the nick of time paid off well over the course of the DIPLOMA project. Outside of research, Jason was always the go-to guy for any hands-on work including soldering a USB adaptor, picking a cabinet lock, and planting flashing LEDs as a prank.

HaoQi Li was the reason we managed to submit the DIPLOMA paper with extended results. In a short period of time, she came up with a working implementation of the photo sharing app that we used to generate several results in our paper. Debugging concurrent code with her was great fun and we had many a Eureka moment together in the lab.

Pablo Ortiz had a significant contribution in this thesis by implementing the

barrier synchronization algorithm. Working with him, both in class and in research, was fun despite the numerous technical road blocks we ran into. He also took great care in correcting my writing, which has definitely seen an improvement ever since.

My labmates Owen, Suvinay, Tushar, Kostas and Manos were great people to hang out with. Owen made sure all our lab machines were up and running at all times, and provided much needed critical reviews when required. Suvinay was a great squash partner, despite both of us not knowing how to play. Tushar was a great example of combining work and play and made sure I never took life too seriously. Kostas and Manos, as senior graduate students, were great sources of advice on coping with the ups and downs of grad school.

Outside of the lab, Varun was a great friend who lent a patient ear to my rants whenever I needed it. Swarun was always there to remind me of the real world that existed beyond grad school. Murali was the commander-in-chief for the board game nights and taught me the ropes of squash. I still hope I can win a game against him some day.

My parents, grandparents and sister have been a great source of support at all times. My sister made sure I didn't get too bogged down by research. My Mom and Dad, each in their own unique ways, were great sounding boards.

Finally, Tejaswi has stuck with me through thick and thin. She has never doubted me, despite my almost perennial state of self doubt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile applications span a wide variety of domains from productivity apps to games. One possible taxonomy of these applications is to consider the extent of communication with the outside world. At one extreme, there are many mobile applications that are still single phone apps, such as a note editor or a single player game, with no communication with the outside world. These single phone apps are typically tested with an actual phone. When access to a phone is expensive or infeasible, developers typically use software emulators such as the Android emulator [2]. Regardless of the development platform (phone or emulator), testing these applications is relatively straightforward since the environment is self-contained: just locally on a phone.

At the other extreme, there are many applications that make extensive use of a phone's communication interfaces. These applications, usually termed services, typically have a thin client running on the mobile phone, connecting to a server in the Cloud, through the cellular network. Examples include mobile mail and web browser clients. Testing typically involves *separate* testing of the front-end client on an emulator/phone, and the back-end service on a server farm injected with synthetic client queries. Such an evaluation methodology ignores the network's impact on overall service performance. This approach worked well for web services when accesses traversed fast, and highly reliable wired links. On a cellular network, characterised by high and variable latencies [35], such an approach fails. Network latencies as high as several seconds aren't uncommon on these networks.

This problem is exacerbated by increasingly sophisticated mobile software. To offload communication from the slow cellular network which is reaching its capacity, researchers have proposed runtime caching of services on the mobile phone [23, 24], runtime partitioning of client-server code [30, 17, 16, 32] and mobile, distributed services [25]. None of these emerging mobile service architectures can be effectively tested and evaluated with today's software emulators that test computing separately from the effects of wireless networking.

Hardware trends similarly tax mobile app testing. Mobile phones with their array of sensors have led to mobile applications and services that are increasingly interactive, such as multi-player online role-playing games, and intelligent transportation services. User interactions, whether explicitly through the touch screen, or implicitly through sensors such as the GPS and gyrometer which track the user's movements, critically impact the application performance. With phone processors rapidly doubling their core counts (The Samsung Galaxy III already has a quad-core processor), it is becoming increasingly difficult to test and project an application's performance on a next-generation phone platform.

A natural solution to many of the above problems is real-world field tests. This option is adopted by most researchers in mobile software. Unfortunately, real-world deployments are logistically difficult to scale beyond the low tens. They also suffer from non-repeatablity : Node mobility, human interactions, network conditions, and sensor values all change from trial to trial. This non-repeatability confounds application debugging since a dropped packet, on any particular run, may reflect the application's inability to process packets fast enough, congestion at the MAC layer, or simply be the effect of a wireless device moving out of range into an area with low or no connectivity. Any or all of the above problems can show up during a trial, and non-repeatability prevents us from isolating what the core problem really is.

In summary, we need an evaluation platform with several use cases :

1. **Performance Evaluation:** The evaluation platform can serve as an important aid in the design and development of a mobile application by estimating preliminary performance results for applications. This would be an important

pre-cursor to deploying an application into the wild, since, initial test deployments are limited in their scale.

2. **Improved debugging:** The tool would allow the application developer to isolate bugs in an application from errors in the operating environment. In real world evaluations, this is impossible. Subsequent real world tests are distinct from each other, and hence it is difficult to control for errors caused by the environment.

3. **Next-generation hardware:** The tool can evaluate a new hardware design's impact on current apps, as well as new app optimizations targeted for future hardware.

4. **Human Interaction:** In conjunction with models that predict user behavior, the tool can help predict the impact of a new application on its targeted user base.

We use the term Mobile Distributed Applications to broadly capture all mobile applications that involve a non-trivial amount of networking. This thesis takes the first steps towards designing a software infrastructure, MobiTest, that allows us to simulate Mobile Distributed Applications. We discuss and document the various designs that we tried and the problems we faced in each of them.

Chapter 2

Design

This chapter presents the design of MobiTest. It documents the various designs we came up with, their problems and some solutions. We also mention problems that we could not solve effectively.

This chapter is organized as follows. We first describe the system architecture at an abstract level 2.1. This system architecture description is characterized by three important functionalities : simulating communication 2.1.1, simulating computation 2.1.2, and ensuring that the clocks of all simulated entities are in sync with each other 2.1.3. First, we look at the module simulating communication in more detail since it is common to all our designs. Second, we focus on the module simulating computation. Lastly, we describe the issue of synchronizing clocks among various simulated entities. In this section, we also describe how the issue of clock synchronization is tied to the process of parallelizing a simulation.

Section 2.2 presents each of the individual designs we came up with along with their attendant accuracy-simulation speed tradeoffs. For each of these designs, the presentation is in terms of the compute simulation, communication simulation and clock synchronization methods. We also include Table 2.1 that compares all our designs in terms of how they realize each of the three components in Figure 2-1.

Finally, after describing the design of MobiTest, we focus on two aspects that are important in simulating any mobile distributed system : sensory inputs 2.3 and human interaction 2.4. We describe how we addressed these issues and the challenges

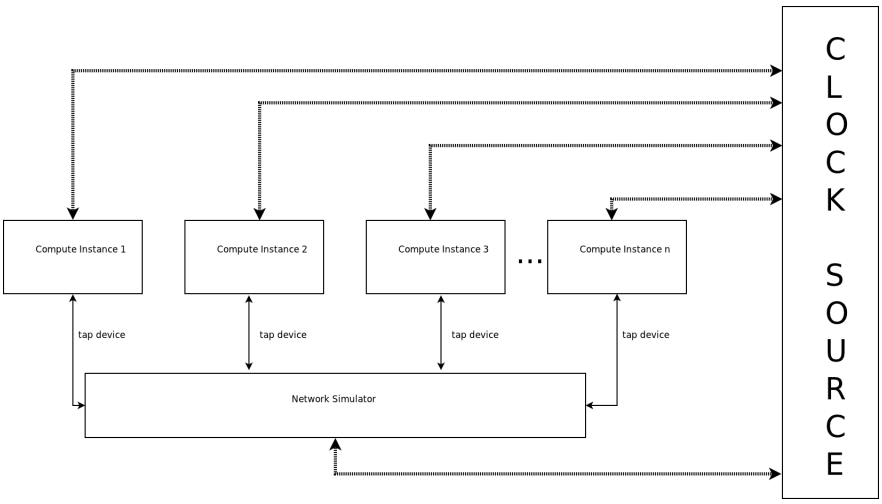

Figure 2-1: Abstract block Diagram of Evaluation Infrastructure

we faced.

2.1 System Architecture

Any evaluation system that deals with simulating distributed systems must specifically address three questions :

1. **Communication Simulation** : How do we simulate the communication between simulated nodes ?

2. **Compute Simulation** : How do we simulate the individual computation that goes on at each of the nodes of the simulated system ?

3. **Clock Synchronization** : How do we ensure that all simulated entities share the same notion of time to preserve simulation fidelity?

An abstract schematic of this architecture is shown in Figure 2-1. We will refer to this architecture while describing concrete designs in Section 2.2. We describe each of the three questions above in turn.

2.1.1 Simulating Communication

Simulation is a well established technique in the field of networking. Several network simulators such as ns-2, ns-3, Qualnet, Opnet and Glomosim [7, 37, 15] exist and are widely used. Most of these simulators support an emulation mode which allows the simulator to be interfaced with another software that simulates the computation. This software is typically a Virtual Machine (VM) that houses the code of interest that needs to be simulated. These VMs could be full blown like VMWare or light weight such as BSD jails or Linux Containers. The only requirement is that the VMs support a network interface. This network interface is connected to a virtual network interface on the host machine. The virtual network interface on the host is known as a tap device.

The network simulator on the host machine reads from this software network interface or tap device. Once it reads packets off the tap device, it can route them appropriately (through the simulated network topology in the network simulator) to other tap devices. These other tap devices are in turn connected to other VMs.

Note that this modularity between simulating communication and computation is desirable so as to allow either component to be replaced without affecting the other. In particular, the network simulator is only aware that there is a tap device on the host machine that can continuosly generate packets. It is agnostic to what generates those packets. It could be a VM, an emulator, a full system simulator or even just a simple packet generator.

In our case, we use ns-3 [7] as the preferred network simulator since it is open source in addition to supporting tap devices. Though other simulators such as Qualnet and Opnet also support connections to tap devices, we decided to use ns-3 since it is open source. ns-3 is a packet-level network simulator, a highly improved and revamped version of the widely used ns-2 simulator. It includes support for various application layer, transport layer, and network layer protocols and is used to simulate a wide variety of wired and wireless scenarios. Specifically, for wireless scenarios ns-3 allows the propagaton model, propagation delay and WiFi transmit power to be configured.

In our case, we set the propagation model to be the standard free space propagation model where the power of the signal falls off as $\frac{1}{r^2}$. The propogation delay assumes speed of light delays for actual over-the-air transmissions and WiFi defaults for other MAC protocol overheads such as SIFS and DIFS. The transmit power is set to 33 dbM to mimic DSRC [5] radio power levels.

2.1.2 Simulating Computation

Now, we look at the other module from the architectural diagram in Figure 2-1, the *Compute Instance* in greater detail. Further, this is the module in which our iterative designs differ : problems in one design leading to the next design. Before diving into specific designs, it is useful to describe the general process of simulating a guest on a host.

2.1.2.1 Emulating a guest on a different host

Our goal is to emulate one architecture (the guest) on top of another (the host). The guest, in our case, corresponds to the architecture of the mobile phone processors, while the host corresponds to the architecture of the workstation machines used for running our simulation infrastructure. Specifically, the guest architecture is ARM, the dominant mobile architecture. Popular phones such as the iPhone, Galaxy S, Galaxy Nexus, and HTC Thunderbolt use the ARM architecture. The host architecture is x86 since work station machines typically use the x86 architecture. There are multiple ways of running a guest architecture on a host architecture when the guest and the host are different and we discuss them next.

The simplest way of running a guest on a host is to use an interpreter that processes the instructions of the guest, one at a time, based on their opcode. Processed instructions are dispatched to a specialized function that executes the opcode. The advantage of such an approach is its simplicity and flexibility. It allows us to "zoom" in on an individual instruction and implement it in as great detail as desired. For instance, let us consider an addition instruction that takes two operands, a and b,

as arguments. If we were simulating a single cycle non-pipelined guest, this could be implemented by an instruction that simply adds the two numbers and returns the sum as such. If the goal were to simulate a more complex out-of-order or pipelined processor, then this implementation of the addition instruction could be replaced by a complex function that models the effects of delays and stalls in individual stages of the pipeline. This function could also model additional simulated hardware such as a reorder buffer. The approach outlined above is the one that is followed by full system simulators such as GEM5 [14] and SIMICS [26]. This approach allows us to simulate hardware at as much detail as we wish, but comes at a cost: it is very slow. For instance, the time taken to boot a shell on such full system simulated platforms is of the order of a few minutes. Section 3.1 has a more detailed overview of GEM5.

The other approach to running a guest on a host is compiling the binary of the guest architecture into one for the host. Compilation can be done statically, and the resulting binary can run at close to native speeds on the host (or faster if the host is a more capable architecture than the guest). However, statically compiling the entire guest binary into a host binary prior to execution is problematic. In particular, it doesn't allow the user to dynamically insert and execute new guest code at run time. This need arises while running a full blown OS/application stack on the emulated guest. In such scenarios, it is typical to install new applications on the guest at run time. To alleviate this, the code is typically compiled at run time using dynamic binary translation. This Just-in-Time approach is used in several Application Level Virtual Machines such as the Java Virtual Machine. Instead of processing each opcode from the guest architecture as it comes and dispatching it to a function handler (as done in an interpreter), the Java runtime compiles incoming bytecodes at runtime into the host instruction set.

The translation is typically done at the level of basic blocks. These are blocks of straight line code that are terminated by a branch or a jump statement. Further, since certain code paths may be frequently used, these translations are stored in a cache. On a subsequent translation, these code paths may just be looked up from the cache and their translations could be reused. Thus, caching allows us to compile a

basic block once and run it multiple times. This can demonstrate great performance benefits for hot code paths such as frequently executed loop bodies. The overhead of compiling itself is typically greater than the overhead of function execution that exists in interpreted simulators such as GEM5 and SIMICS. However, this overhead is amortized over several repeated executions of hot code paths. In contrast, in a full system simulator, the function execution overhead is incurred on every instruction.

2.1.3 Clock Synchronization & Parallel Simulation

We have just outlined the general procedure for simulating computation and communication. Now, we can treat the network and each of the individual nodes (running their own software) as separate entities in a simulation. It is entirely possible to run such a simulation on a single thread with one event queue. In such a scenario, clock synchronization is easy to achieve because the single thread keeps track of all events and hence there is only one global clock. In fact, such an approach is much easier to design and build. However, this doesn't scale well with the number of simulated nodes. Particularly, it takes no advantage of growing core counts on workstation machines.

Parallel Simulation is one solution to this problem . However, amongst all parallel applications, simulation is notoriously hard to parallelize [33]. This is because events can be simulated concurrently only if they don't have a causal dependence on each other. In other words, clocks between all simulated entities must strive to stay as close to each other as possible. There is only one scenario under which it is acceptable for simulation clocks to drift. This is when it can be guaranteed that the two concurrently simulated events are causally independent of each other.

Failure to respect causal dependence results in loss of accuracy. This loss of accuracy needs to be estimated. Causality means the following. B is said to be causally dependent on A if, by the semantics of the simulated world, event A's occurence potentially has some effect on the occurence or resulting outcome of event B. With this definition of causality, parallel simulations typically take one of two approaches to improve performance :

1. They can be conservative by choosing to simulate exactly those events concurrently that are guaranteed not to be causally dependent.

2. Alternatively, they can be opportunistic, where events are simulated concurrently with the assumption that they are causally independent, but a rollback mechanism exists to undo any effects of causality violation.

Implicit in both these approaches is the assumption that the simulation engine can decide which events are causally dependent on each other. Despite the inherent performance gains, neither of the above two approaches may be feasible in a real system. It may not always be possible to infer causal dependence since it is intricately tied to the nature of the simulated system. To address this issue, we use barrier synchronization to achieve scalability with an increasing number of simulated nodes. We describe this next.

2.1.3.1 Scaling Parallel Simulations : Barrier Synchronization

Barrier synchronization [28] is used to scale parallel simulation of multiple simulated instances. The approach works as follows :

Every simulated entity i.e. either the network simulator or any of the simulated nodes simulates a given amount of simulation time. After this simulation time (called the simulation window) expires, each simulated entity has to wait till every other entity simulates the same simulation window before moving on to the next window.

Note that there is no mention of causality here. So, within a simulation window there may or may not be a violation of causality. However, it should be intuitively clear that the smaller the time window, the less likely this is to actually occur. This is because the clock drift between two simulated entities is limited to the size of the time window. This technique effectively sidesteps the issue of causality which is impossible to compute in some cases. Since barrier synchronization occurs periodically after every simulation time window, the choice of time window is critical. The interval must be small enough to be accurate while being large enough to not exacerbate the simulation time. We use an efficient barrier synchronization algorithm proposed

by Mellor-Crummey and Scott in [28] to ensure that the barrier is scalable. The algorithm is a tournament synchronization algorithm in which the fewest possible number of messages are exchanged to advance the algorithm. The synchronization tree is treated as having different fan-outs depending on whether the synchronizing nodes are all in the process of reaching the barrier or in the process of leaving it. These fan-outs are picked in such a way that the shortest critical path to reach a barrier or to resume computation is acheived. We chose to use this algorithm because its optimizations suited our needs and because it can be analytically shown to scale well with the addition of more nodes. Thus, the algorithm could be expected to perform reasonably well with both our experimental set up and that of others.

Pablo Ortiz, a graduate student at MIT, assisted in the design and implementation of the barrier synchronization especially in choosing the tournament synchronization algorithm.

2.2 Specific Designs

Here, we describe specific design choices for MobiTest along with their attendant accuracy-simulation speed tradeoffs. Wherever possible, we try and relate the design choices to our abstract architectural diagram in Figure 2-1. Table 2.1 compares our design choices in terms of compute simulation, network simulation and clock source. Our target in all cases is to finally simulate a stock Android system, together with off the shelf apps running on top of an Android-supported Linux kernel. The kernel, in turn, would run on a simulated ARM architecture machine. The simulation itself would be carried out on workstation class machines running x86.

2.2.1 Design 1 : Linux Containers + ns-3

Originally, we planned to simply reuse the emulation feature available in a stock network simulator such as ns-3. ns-3 has a real time mode where it can connect to isolated light weight VMs each running their own application and having their own distinct IP address. The connection between ns-3 and these containers is achieved

Section No.	Design Description	Compute	Network	Clock source
Section 2.2.1	Linux containers + ns-3	Linux Containers	ns-3	Wall clock / Real time
Section 2.2.2	Android-x86 + ns-3	Android-x86 on VirtualBox	ns-3	Wall clock / Real time
Section 2.2.3	QEMU + ns-3 + icount + barrier-sync	Android emulator	ns-3	Barrier Synchronization with configurable barrier
Section 2.2.4	QEMU + ns-3 + micro-op + barrier-sync	Android emulator	ns-3	Barrier Synchronization with configurable barrier
Section 2.2.5	QEMU + ns-3 + SystemC + barrier-sync	Android emulator	ns-3	Barrier Synchronization to SystemC time source
Section 2.2.6	GEM5 + ns-3 + barrier-sync	GEM5	ns-3	Barrier Synchronization with configurable barrier

Table 2.1: Comparing Design Choices

using tap devices which are virtual network interfaces, as explained earlier.

The light weight VMs could be BSD Jails, chroots or Linux Containers. No matter what the type of the Light Weight VM, it is important to note that this is a very lighweight application-level virtualization mechanism. VMs are isolated in the sense that each VM has its own IP address and can run its own applications. In contrast, the file system, kernel and networking stack are shared by all the light weight VM instances. The shared file system, kernel and networking stack are the same as that of the host on which all these light weight VMs run.

The "simulation clock" of all these light weight VMs is the same as the host clock, otherwise known as wall clock time. The network simulator may be running faster than the wall clock and might process packets faster than they are generated. Being a discrete event simulator, the simulator would immediately skip time and fast-forward onto the next event in its discrete event queue. However, the light weight VMs might not be at that time yet, since they are all simulating themselves at wall clock time which is a continuous time source.

To alleviate these issues, ns-3 supports a real time mode. In the real time mode, the event time stamps are not merely used for relative ordering in the discrete event priority queue. Their absolute values matter as well. The values refer to the exact wall clock time at which those events should be simulated. This ensures the simulator is never too far ahead of the VMs. Hence, the network simulator sleeps until the event time stamp is reached.

This approach works well when the network simulator is processing packets faster

than they are being generated. If the generation rate is much more than the processing rate, the network simulator will struggle to process packets before the next one is due to be processed. After a point, the network simulator starts missing deadlines and the accuracy of such a system is questionable.

Besides, Linux Containers are unnecessarily restrictive since they imply that the guest and host share both the ISA and the kernel. This precludes the simulation of an ARM architecture on x86, our target scenario. Lastly, Linux Containers are traditional linux systems whose operating environment is significantly different from Android, the predominant mobile phone OS. In particular, Android applications typically do not run as native binaries but as .apk files. Apk files are analogous to .class files which contain Java bytecode, and need the Dalvik application level VM to run. Linux containers run on traditional desktop machines that have no support for the Dalvik VM.

2.2.2 Design 2 : Android-x86 + ns-3

As a next step, we decided to investigate Android-x86 [4] which is a complete port of Android (including the Operating System, libraries, user space applications and Networking Stack) to the x86 architecture. The initial impetus for such a project came from the desire to run Android on netbooks. The same kernel image can also be used to run Android on a full blown VM monitor such as VMWare or VirtualBox. We experimented with this idea and were successfully able to communicate between two such instances using ns-3's real time emulation mode

However, this design suffers from the same real time requirement problem that Design 1 also faced. Ideas like SliceTime [34] get around this problem by using barrier synchronization 2.1.3.1 to synchronize instances repeatedly and removing the real time requirement altogether. However, SliceTime requires changes to the hypervisor to run and runs only on a bare metal hypervisor such as Xen. It isn't readily applicable to hosted hypervisors such as VirtualBox and VMWare .

Lastly, x86 isn't representative of mobile phone architectures. Despite these apparent shortcomings, such an approach is useful in its own right. It would be very

accurate for simulating tablets and netbooks that run x86 processors, and extremely fast if the host x86 workstation machine had hardware virtualization support. This observation is corroborated by the fact that Android recently released a hardware virtualized version of its emulator for x86 architectures [1]. As expected, the hardware virtualization makes such an emulator much faster than using binary translation.

In summary, though Android-x86 serves its own niche pretty well, what we essentially need is a complete port of Android on ARM, which is a substantial effort requiring deep kernel understanding.

2.2.3 Design 3 : QEMU + ns-3 + icount + barrier-sync

Designs 1 and 2 both suffer from one common problem. Whether they run on lightweight VMs (Design 1) or heavy weight VMs (Design 2), they suffer from the real time requirement problem because their "simulation clock" is the same as the wall clock on the host machine. Since this eventually limits scalability, we decided to relax the real time constraint by using the same barrier synchonization technique described earlier (Section 2.1.3.1). In this context, we use barrier synchronization to synchronize among several clocks to make sure they never get too far away from each other. The use of barrier synchronization is common to designs 3, 4, 5, and 6 and the reader is referred to Section 2.1.3.1 for a recap of how the technique works. The rest of this section will focus on Design 3.

The Android emulator is the de-facto emulator that is used for testing all Android applications in a software environment. It is based on QEMU [12], a popular open source emulator ported to support several guest and host architectures. QEMU is an emulator which aims to achieve a high level of performance. To do so, QEMU implements binary translation as described earlier.

To interface with a network simulator QEMU implements a virtual network interface card inside its emulated system that can be linked to a virtual network interface on the host system. This host system interface is the same tap device that ns-3 reads from to implement real time emulation.

Binary translation improves performance substantially but comes at a cost; the

resulting system has no accurate timing characteristics. In other words, though the Android emulator is routinely used for testing functionality, it can't be used for profiling. QEMU does little to model the clock of the simulated system and only tries to maintain it close to the wall clock on the host system. This is prone to error since the wall clock time might have no relation to the virtual time in the simulated system.

Our first design approach was to add accuracy on top of such a system. QEMU provides an option "icount" which uses the instruction count to drive the simulated clock instead of using the wall clock time (which is extremely inaccurate). This option also allows us to specify the clock frequency of the simulated clock as N where 2^N is the number of nanoseconds per instruction [8].

The clock is derived by measuring the number of instructions that have been executed and associating a parameter (specified by the user using the value N) corresponding to the number of executed instructions per unit time. QEMU runs a main loop, where the instruction count is incremented after every iteration in the main loop. Every iteration of the main loop, however corresponds to several thousand executions of basic blocks and each block can span several instructions. We use the clock time at the beginning of every iteration of the main loop to drive the barrier synchronization algorithm described earlier by comparing it to the current barrier window. If the time exceeds the barrier window, the simulation pauses and waits until it gets a proceed signal from the synchronization server. Otherwise, it continues executing the next iteration of the main loop.

This approach turned out to be problematic since every iteration of the main loop corresponds to several thousand instructions (at the very least). This meant that the clock derived from the instruction count at the beginning of each iteration was far from smooth. In contrast, it jumped up by several milli seconds before each iteration of the main loop. This was because several thousand instructions were executed before control returned to the main loop in QEMU. If the barrier synchronization interval was much smaller than the magnitude of these "jumps", the whole point of barrier synchronization would be lost. The clock would at times exceed the barrier by a vast amount at which point stopping and pausing would have no effect.

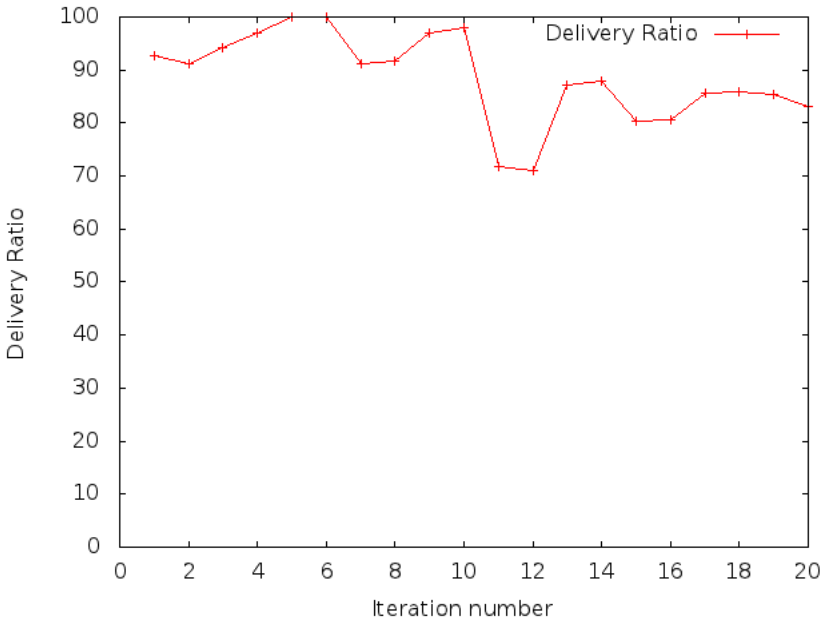

Figure 2-2: Delivery Ratio across 20 iterations for 2 nodes pinging each other

This "jumping" behavior has a marked effect on application level metrics. We ran a ping test between two such instances connected across ns-3. The scenario within ns-3 was setup such that the nodes were static and connected via a wireless link between them. The nodes were well within range of each other as per the 802.11 Wifi standard used in the simulator. Hence one would expect close to constant Round Trip Times, and no losses when two such nodes ping each other. However, what we observed was significant loss rates (the delivery ratios are only 70 in some iterations as opposed to 100) as shown in Fig 2-2. The inability to reproduce such a simple scenario led us to conclude that the modeling of time in the system was flawed because of the "jumping" behavior described earlier.

Although the timing characteristics of this design were poor (as illustrated above), we were able to scale the infrastructure to run on 100 simulated nodes. To simulate a real benchmark which took about 15 minutes in the real world, we took between 1 and 4 hours depending on the exact parameters of the experiment (such as the Wifi transmit power in ns-3 which in turn influenced networking activity). Figure 2-3 shows the parallel speed-up achieved by Design 3. The speed-up is plotted as a

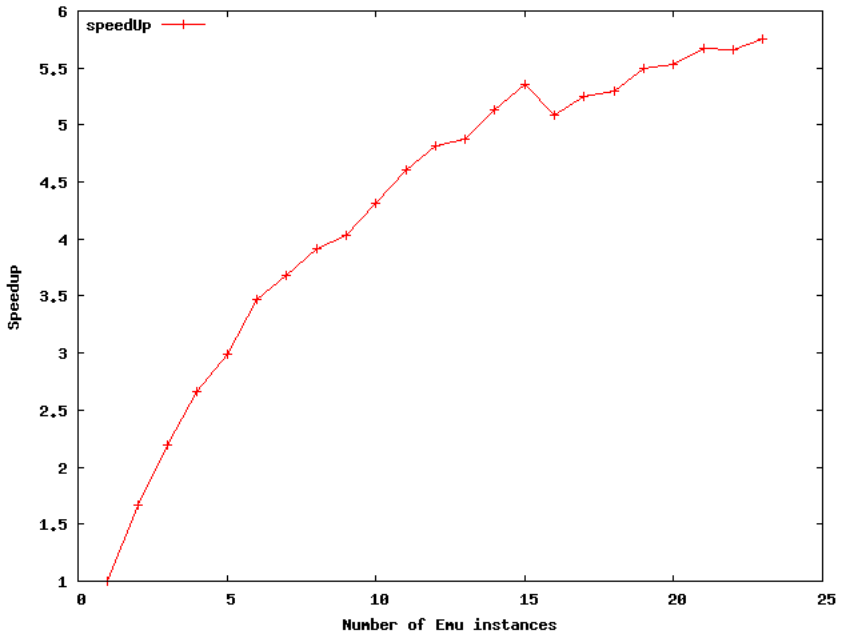

Figure 2-3: Speed-up as the number of QEMU instances is varied from 1 to 24

function of the number of nodes being simulated. The simulation kernel is a simple networked microbenchmark that runs for 9 seconds of real time. The speed-up is calculated as described next.

We measure the time taken to run the microbenchmark when the number of nodes N is 1. We multiply this number by N to estimate a *hypothetical sequential simulation time* which gives the simulation time of a purely sequential implementation for any N. Let's call this $seq(N)$. Then, for a given N, we measure the actual simulation time $sim(N)$ using Design 3 and its associated barrier synchronization algorithm. We compute $\frac{seq(N)}{sim(N)}$ as the speed-up and plot it as a function of N. We see that the speed-up is far from being linear, but we do get substantial speed-up with increasing N. This justifies the use of barrier synchronization as an effective means of parallelising our simulation.

In summary, Design 3 achieves scalability and functional simulation while failing to model the timing characteristics of the system accurately.

2.2.4 Design 4 : QEMU + ns-3 + IR + barrier-sync

The problems with the above design led us to the next design. From what we observed, the cause for the bad correspondence between the real world and the simulated world was the "jumping" effect due to the coarse nature of time in the simulator. Before describing our next design, a discussion of QEMU's internal design for binary translation is in order.

To achieve portability across several different host architectures, QEMU uses an intermediate representation during the binary translation process. This intermediate representation (IR) consists of *micro-ops* in an idealized machine language that doesn't conform to any particular host ISA. The IR allows QEMU to be designed by just translating between the guest ISA and the IR. This portion of the code is common to all host architectures. The IR also allows QEMU to carry out traditional program optimizations like dead code removal. The IR, in turn, is compiled into an object file that contains the relevant host machine instructions to implement each of the micro-ops in the IR. This compilation from IR description (consisting of micro-ops) to the object file (with the host machine set instructions) is carried out by a compiler for the host instruction set. Thus QEMU itself doesn't need to concern itself with portability and can deal with just the translation into the IR which is common across all architectures. The IR to host machine code translation is carried out by the host compiler which is likely to exist for other purposes anyway. Hence the portability concern is shifted from writing translation backends for every host ISA (a much harder task) to ensuring that there is a compiler for each host ISA. The latter is a much easier task that is almost taken for granted).

The IR also provides a convenient way to instrument code. When translating from the guest machine code to the IR, a given guest machine opcode can be translated to one or more micro-ops in the hardware. Thus if it is required to count every single instruction ever executed, every translation from guest machine code to micro-ops can also generate a "counter micro-op" that simply increments a global instruction count. More accurate counting can also be achieved by populating a look up table

that maps from instruction op code to number of clock cycles per instruction. With this lookup table, the "counter micro-op" can count cycles instead of instructions and can attribute a different number of cycles per instuction to different opcodes. This cycle count can in turn, be used to drive a clock, which in turn can be used for barrier synchronization. This approach is an oversimplification, since it assumes instructions follow each other serially and precludes out-of-order execution and pipelining. However, it is a reasonable first cut model to measuring time in a smooth manner.

We constrast this to the 'jumping' problem earlier where time jumps of 10 ms were common since instuction counts were incremented in several thousands. This jumping was because several thousand basic blocks were executed before control returned to the main loop. In contrast, we increment clock cycles at the level of each individual instruction. Since an instruction on ARM is typically no more than 4 clock cycles, this corresponds to a granularity of 4 clock cycles or 4 ns at a 1GhZ simulated clock which is much smoother compared to the 10 ms jumps we saw earlier.

We further enhance these counter micro-ops to not just count cycles, but also use these cycle counts to implement barrier synchronization within the micro-ops body. Hence these counter micro-ops check the number of elapsed clock cycles against the barrier target at that point in time, and appropriately stall QEMU if the barrier has been executed. Since the clock's granularity is much smaller (4 ns), fairly small barrier windows can be used, which provide us with better accuracy. These counter micro-ops are integrated into the dynamic translation process by generating a counter micro-op for every guest instruction that is compiled into micro-ops. This ensures that every instruction is counted.

This better accuracy does come at the cost of performance though. Since the translation for every guest instruction now also includes the call to a micro-op to increment a counter, there is barrier synchronization overhead incurred on every guest instruction as opposed to an overhead once every iteration of the main loop (which was the case earlier). This design provides a highly accurate clock and is similar to the approach followed by QEMU's trace option which is used for profiling. Two other projects [20, 29] implement similar ideas. However, neither is in active use and both

are designed for simulating embedded systems in specific domains and not Android applications.

Having described the merits of this design , we now proceed to describe one pitfall with the design that necessitated a rethink of the design. We describe that scenario next. ARM has certain instructions such as Wait-For-Interrupt(WFI) which halt the clock and then put it into a low power mode until an interrupt fires. This interrupt is then used to wake up the clock again. The interrupt could be external or a timer driven interrupt. Let us consider the case of a timer driven interrupt. In this case, the CPU stops until the timer interrupt fires to wake it up. The timer interrupt fires after a specific period of time elapses. However, in our system we use instruction counts to increment time and drive the clock. In other words, time can't progress or elapse until instructions execute that increment time. Yet, the CPU has been stopped by the WFI instruction until the specified time has elapsed. This leads to a deadlock, where the entire system stalls without making any progress.

The problem doesn't arise with using the wall clock time as the reference because the wall clock time passes all the time anyway, and isn't tied down to the execution of instructions. This suggests that any sensible way of measuring time ought to be decoupled from the execution of instructions itself, because time can and should pass with no instructions being executed.

2.2.5 Design 5 : QEMU + ns-3 + SystemC + barrier-sync

The need for an independent reference source for monitoring time prompted us to revise our design again. This time we considered using SystemC, a system description language modeled after C. SystemC also has a simulation engine, similar in spirit to the simulation engine in Verilog, that allows time to progress on its own. There is also an extension of QEMU that allows it to interoperate with SystemC [29]. In this extension, SystemC is used to drive the time reference in QEMU. However, this system is targeted at specific hardware platforms and has very limited user support or support for Android.

Furthermore, there are very few standardised SystemC implementations of the

peripherals that SystemC intends to model. So, although SystemC was designed to be a high level system description language with the capability of modeling different hardware, reality is significantly different. In practice, very few standardized descriptions seeem to exist. Hence, adopting this design requires designing a SystemC model of an ARM system from scratch.

2.2.6 Design 6 : GEM5 + ns-3 + barrier-sync

The problems with modeling time in QEMU led us to using a full system simulator that explicitly models time as an independent entity by itself. Specifically, we used GEM5 [14], an existing open source full system simulator that is widely used in the architectural community to evaluate the impact of various micro-architectural designs on the performance of benchmark applications.

GEM5 is a discrete event simulator that can model systems at varying levels of detail. For the same instruction set architecture, GEM5 can simulate a functional model, a timing based model, a pipelined model that is strictly in-order or an out-of-order execution model. The more detailed the model, the longer the simulation time. GEM5 has a main loop that picks the next event off the event queue and simply executes that event, and then removes it from the queue. This continues until the queue is completely empty. These events could be compute related, such as an opcode execution, or IO-related such as servicing a hardware interrupt. Models for hardware peripherals (device controllers) are written in C++, just like the core of the simulation engine itself. At the beginning of every iteration of the event loop, we use the barrier synchronization algorithm to check the clock against the current value of the barrier window. If the barrier window has been exceeded, the simulation pauses until it gets permission to proceed again.

GEM5 connects to the Network simulator ns-3 through tap devices just like QEMU and all our earlier designs. We will detail this design in Chapter 3

2.3 Simulating Sensors

Mobitest is designed to simulate distributed mobile applications. Almost every phone today has sensors of some kind and many of them have several sensors with very different characteristics. It is important that sensory inputs to a system be simulated correctly for the system to be useful.

The level of support for sensory inputs depends on the support for the respective device drivers built into the kernel of the simulated node. For the Linux Containers based design, there is no such support simply because the kernel is a standard Linux kernel for desktops that do not have any such sensors. The QEMU based solutions do have extensive support for simulated sensor input. The Android emulator in this case supports sensor based input by allowing a terminal to interface with the running simulator by sending it location fixes and accelerometer and gyrometer readings. The GEM5 based systems have little or no support at present. However, as mentioned earlier, such support can be built in by compiling the appropriate device drivers. For instance, the current version of Android-x86, which is itself an Android port for the x86 architecture supports the webcam.

For sensors such as the camera and the microphone, a work around could be feeding the simulated node a set of stock images or audio samples for analysis by the algorithm running on the simulated node.

2.4 Human Behavior Modeling

One of the potential use cases for such a system is human behavior modeling. Put differently, here we try and evaluate what impact a new application would have on its real users i.e. we focus on user level metrics instead of simply application performance. As a concrete example, consider the case of Intelligent Transportation Systems (ITS) applications. Such apps leverage the technology available on smartphones and road side sensors to make driving safer and more efficient. Typically they notify the user through a pop up or audio message which the user then responds to. The response

could result in a change of route, braking, stopping or simply reducing the vehicle's speed. Examples of such applications include SignalGuru [25], a Green Light Advisory service; ParkNet [27], a crowdsourced service that determines parking lot occupancy, the PotHole Patrol [18], and Surface Street Traffic Estimation [36]. In terms of our simulation framework the effect on humans and their resulting response would result in the change of some of the simulated sensory input. As such, there are two ways of simulating such interactions.

1. Open Loop : Here, the sensory inputs are all know apriori like a sensor trace of a car's drive or a pedestrian's commute. The trace is used to simulate sensory inputs at various points in simulated time and hence affects any portion of the application that depends on this simulated sensory input. In effect, this is the same as collecting a trace file and passing it as a parameter to the simulation when it begins.

2. Closed Loop : Here, only the sensory inputs at the beginning of time are known when the simulation starts. Starting from this set of initial conditions, the sensory inputs vary in response to the actions taken by humans to various outputs from their ITS Apps. TraNS [31] follows this approach. This closes the loop and can truly evaluate the effect of an ITS App on usable metrics that users care about.

In our case, we attempted to integrate our simulation system MobiTest with SimMobility [9], a state of the art simulation platform to simulate transportation systems at various levels of granularity ranging from a few milliseconds to days to several years. We are particularly interested in the short term or ms-by-ms simulations. SimMobility would feed sensory inputs into our system, which would act on those and produce outputs from the ITS Apps. These outputs would in turn influence behavioral models that exist within SimMobility, which in turn will change the position, acceleration, photo views and orientation readings. This will, in turn, feed back into MobiTest if the system were closed loop.

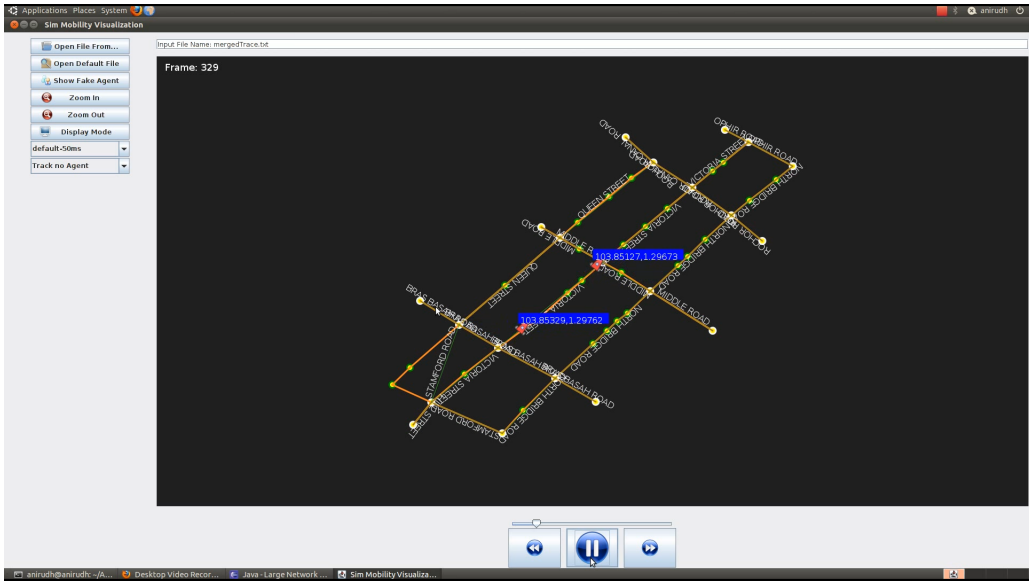

Figure 2-4: SimMobility visualizer visualizing output from Sonar app

In our case, at present, we have an open loop system where the sensory traces are derived by running SimMobility first, followed by collecting a trace from SimMobility's output. This trace is then fed into MobiTest as a sensory input which in turn drives an ITS Application running inside MobiTest. However, we could achieve this integration only with the QEMU-based Android emulator designs, since none of the other designs have support for sensors yet. As a demonstration of this open loop control, we ran a simple sonar app on the simulated nodes which would broadcast its own node's GPS coordinates. Any node that hears another node's broadcast would display the other nodes' coordinates on its own screen. We use the same visualizer as that used for SimMobility to visualize the output from the sonar app. We show a screenshot in Figure 2-4. In the future, once the loop is closed we should be able to carry out more accurate human behavioral modeling studies. This integration with SimMobility was carried out with assistance from Seth Hetu, a graduate student at the National University of Singapore, who is one of the lead developers of SimMobility.

Chapter 3

Implementation

3.1 Overview

Given the pros and cons of the various designs outlined in the previous chapter, we decided to use Design 6 as the basis for further implementation and we describe it in greater detail below. This design uses GEM5 as the computation simulator and ns-3 as the networking simulator.

This chapter is organized as follows. We first describe the overall architecture of the GEM5 simulator. Second, we describe the changes we had to make to support barrier synchronization among GEM5 instances. Third, we describe the changes required to interface a single GEM5 instance to ns-3 using a tap device. Fourth, we describe our experiences in making these changes on an ARM version of the GEM5 simulator. Lastly, we describe our experiences in making these changes on an Alpha architecture version of the same simulator.

3.2 GEM5 Primer

GEM5 [14] is a popular open source simulator that is commonly used in the Computer Architecture community. Typically, GEM5 is used to modify and evaluate the Network On Chip topology, microarchitecture, cache sizes and memory access latencies of the simulated hardware. These changes in turn, affect the performance of the archi-

tecture on standard benchmarks such as the SPEC [11] and PARSEC [13] benchmark suite. GEM5 helps measure the impact of these microarchitectural changes and collects statistics on metrics of interest while changing a computer's microarchitecture. These metrics could be application runtimes, memory access latencies, cache hit and miss rates, number of pipeline stalls and so on.

GEM5 is predominantly used to simulate single CPU instances although each CPU instance can itself simulate multiple cores. In our case, our scenario of interest is a multi-CPU simulation where each of the CPUs could itself be composed of several cores. Current support for such *networked simulations* in GEM5 is somewhat limited. At present GEM5 only supports an ethernet link between two machines. There is no support for wireless networking and broadcast.

GEM5 also has several levels of detail in its simulation. A *functional* model simply simulates the functionality or correctness of a target architecture. A *timing* model attributes a variable number of clock cycles to each instruction and differentiates between different instructions' processing time using a lookup table. An *in-order* model simulates an in-order, pipelined CPU and an *out-of-order* model simulates further concurrency by allowing instruction reordering. The least accurate model i.e. the *functional* one is the fastest in terms of simulation time and vice versa. GEM5 also allows checkpointing where the current state of the simulation can be suspended and captured to be resumed later. Additionally, it supports sampling which is selective execution of an application to speed up performance. Checkpointing and sampling can be applied to any of the above models, and hence can improve the simulation speed of MobiTest as well.

Since it is designed as a full system simulator, GEM5 is typically used to boot an entire operating system image targeted for a particular architecture. The OS image contains the kernel binary which contains all the code to execute the kernel. It also contains the file system image which stores an exact copy of user space binaries as they would appear on the hard disk of a real system. In fact, this file system image can be mounted as a virtual disk partition on a Linux machine. The file system image contains the benchmarks that the user intends to evaluate in the form of native

binaries. In the case of Android, the file system would include not just the Android binaries which are written in a high level byte code, but also the Dalvik runtime, which is the application level virtual machine that runs Android bytecode files.

3.3 Implementing Barrier Synchronization in GEM5

GEM5 runs a tight event driven loop where it processes all events one after the other until it runs out of events in the priority queue. The high-level code (reproduced from gem5/src/sim/simulate.cc) for this is given below :

```
while(1)    {
       // there should always be at least one event (the SimLoopExitEvent
       // we just scheduled) in the queue
       assert(!mainEventQueue.empty());
       assert(curTick() <= mainEventQueue.nextTick() &&
              "event scheduled in the past");
       // forward current cycle to the time of the first event on the
       // queue
       curTick(mainEventQueue.nextTick());
       Event *exit_event = mainEventQueue.serviceOne();
  }
```

We modify this code to perform barrier synchronization as follows :

```
while(1)    {
       // there should always be at least one event (the SimLoopExitEvent
       // we just scheduled) in the queue
       assert(!mainEventQueue.empty());
       assert(curTick() <= mainEventQueue.nextTick() &&
              "event scheduled in the past");
       //----------Barrier Sync-----------------------------------------------------------
       //check to see if enough time has passed for another synch to take place
       if(atoi(simulationMode) == 2)
       {
          now.setTimer();
           if((curTick() - start_tick) >= (num_syncs * syncInterval))
           {
               //perform barrier synchronization
               bool ack=false;
               while(!ack) {
                 SendReadySignalToServer(socket_file_descriptor);
                 ack=GetAckFromServer(socket_file_descriptor);
               }
```

```
            ++num_syncs;
        }
    }
    //----------------End of Barrier Sync ----------------------------------------------------
    // forward current cycle to the time of the first event on the
    // queue
    curTick(mainEventQueue.nextTick());
    Event *exit_event = mainEventQueue.serviceOne();
}
```

Here, start_tick is the number of CPU ticks at the beginning of the simulation which defaults to 1. Thus, the code checks every instant that the current barrier has not been exceeded and if it has, it simply waits on an ACK from the server. The tight while loop to check for the ACK is due to TCP semantics. If the barrier sync server dies for any reason, the blocking call to GetAckFromServer terminates immediately with a broken pipe error. This is despite the fact that no data has actually been received. Hence, simply blocking for an ACK using GetAckFromServer is not sufficient.

3.4 Interfacing GEM5 with ns-3

Since we use ns-3 as the networking simulator we must be able to interface GEM5 with ns-3 in some manner. ns-3 supports a real time emulation mode using tap devices as described earlier 2.1.1. In this mode, ns-3 reads from a tap device that the computation entity writes into. Similarly, if the computation simulator wants to read data, ns-3 raises an interrupt on the tap device to the computation simulator every time ns-3 writes data into it.

The mechanism above describes the case where ns-3 is running in real time mode. However, we described the shortcomings of the real time mode earlier. Since we use barrier synchronization we need to perform a similar sort of barrier synchronization in the ns-3 event scheduler. For this, we use the implementation of barrier synchronization for ns-3 provided in [34].

To interface GEM5 with ns-3 we ensure that all writes into the simulated network device reach a tap device on the host system. To do this, we leverage the EtherTap

module provided within GEM5. The EtherTap module opens a TCP listening socket on a particular port. One end of the EtherTap module is connected to the simulated network device, such as the Intel E1000 for which the device controller is written as part of the GEM5 source code. The other end of the EtherTap module is connected to the TCP listening socket. The job of the EtherTap module is to forward packets between the simulated network device and the TCP socket. However, this isn't sufficient. We need to be able to pass the packets that appear on the TCP socket into a tap device. For this purpose, we use socat [10] which is a multi-purpose relay that allows the user to connect two end points of almost any type together. In our case, we use socat to connect the TCP socket with a tap interface(A) so that any packet on the tcp socket appears on the tap interface and vice versa. As a last step ns-3 reads from a tap interface of its own. Here, we use socat for a second time to connect the GEM5 tap interface (denoted as A earlier) to ns-3's tap interface. This allows us to move packets freely between ns-3 and GEM5.

3.5 Making the changes to an ARM simulator

GEM5 supports multiple target architectures including ARM, Alpha, x86 and MIPS. Since mobile distributed applications are our target scenario for MobiTest we initially tried to implement the above changes on a version of GEM5 that simulates the ARM ISA. Further, to more accurately model our target scenario, we wished to run Android on this framework. This turned out to be problematic for two reasons :

1. The EtherTap module works by connecting the simulated network device to the TCP socket. Data can appear at the simulated network device only if there is a means for applications to write into it. An application can write into a Network Interface Card only if the kernel exposes the same as an interface (i.e. the interface shows up on an ifconfig). The current version of OS kernels available for ARM do not support an Ethernet Interface. Later, during the course of our work, a version of ARM that supports a PCI-based NIC (the Intel E1000) was released by Ali Saidi of ARM R&D, Austin, TX. However, we were still

unable to use this kernel because of error-prone interactions between the device controller for the Intel E 1000 within GEM5 and the EtherTap module. This setup, however, does work with GEM5's own internal network link (EtherLink) module.

2. Even after the first issue is resolved, for an ARM based system to be viable and widely useful, it would be important to port an Android based system onto it. Porting Android on to new hardware isn't as simple as adding a new file system with the appropriate binaries to a stock Linux Kernel. In other words, the Android framework encompasses not just user space binaries and libraries but also an Android specific kernel. The changes that the Android kernel makes to the stock Linux kernel are modest, totalling about 25000 lines in a 1 million line code base [3]. However, the two kernels have their own separate source trees and hence the changes aren't consolidated anywhere. This makes it hard to add Android support to a stock Linux kernel even assuming the stock kernel already takes care of supporting an Ethernet device.

In summary, the barrier synchronization enabled system doesn't work yet on an ARM-based architecure because of the above issues. An Android kernel image that includes an Ethernet interface will have to be created from scratch for GEM5, a significant development effort.

3.6 Making the changes to an Alpha simulator

In view of time constraints, we moved to making the same changes to an Alpha based simulator, based off GEM5, where we were more successful. The obvious shortcoming of using an architecture such as the Alpha is that of porting Android on to it. However, we decided to use Alpha, since it was the only available architecture that would allow us to demonstrate that a barrier synchronization based approach works in practice.

The Alpha kernel, which is part of the GEM5 distribution, already has support for Ethernet built into it, thereby solving issue 1 outlined above. We were able to

successfully integrate barrier synchronization into the Alpha simulation framework. In the process, we ran a few simple benchmark applications that demonstrate the utility of barrier synchronization enabled simulation and we describe them next.

Chapter 4

Evaluation

This chapter presents results from the evaluations we carried out on the Alpha architecture based simulation infrastructure that we developed.

4.1 Accuracy-Performance tradeoff

One of the benefits of using barrier synchronization is the flexible mechanism to tradeoff accuracy for performance or vice versa. We try to explicitly quantify this. The test benchmark we use is a simple ping test where two simulated machines ping each other 25 times over the simulated ns-3 network. Our baseline or calibration reference is the measurement we get by running the same benchmark over GEM5's own internal network link model i.e. EtherLink. We use this as our baseline since we don't have access to real Alpha hardware. We feel this is reasonable, because the EtherLink model runs a completely deterministic simulation. Both simulated systems (at either end of the EtherLink) add and remove events from one shared event queue and there is no chance of clock drift between the simulation clocks of both machines.

We vary the barrier synchronization interval in factors of 10 from 1 μs all the way to 10 seconds. We measure both the simulation time (performance) and the mean Round Trip Time (RTT) reported by the ping benchmark inside the simulation (accuracy). The performance and accuracy graphs are both shown below in Figures 4-1 and 4-2.

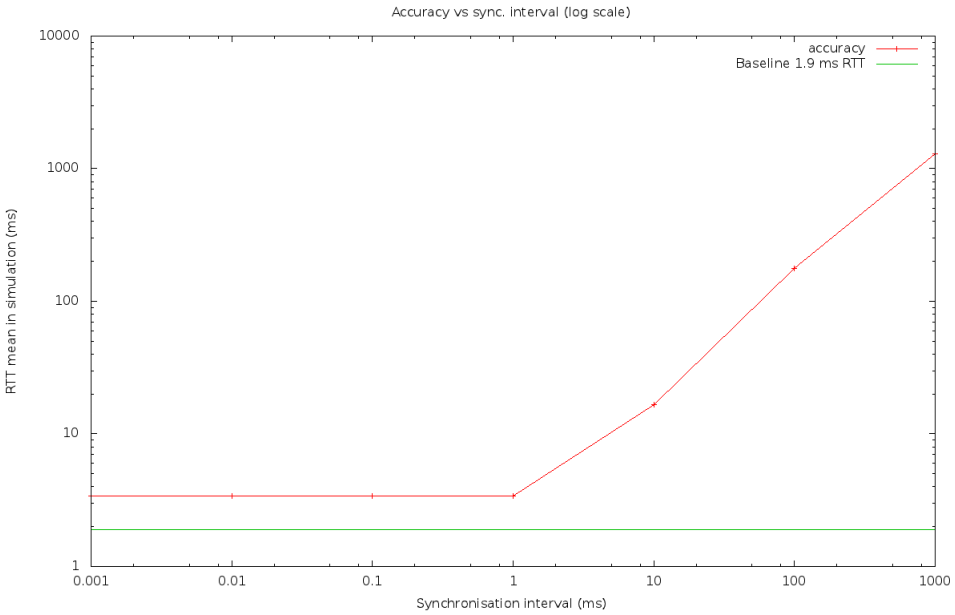

Figure 4-1: Accuracy of Mobitest

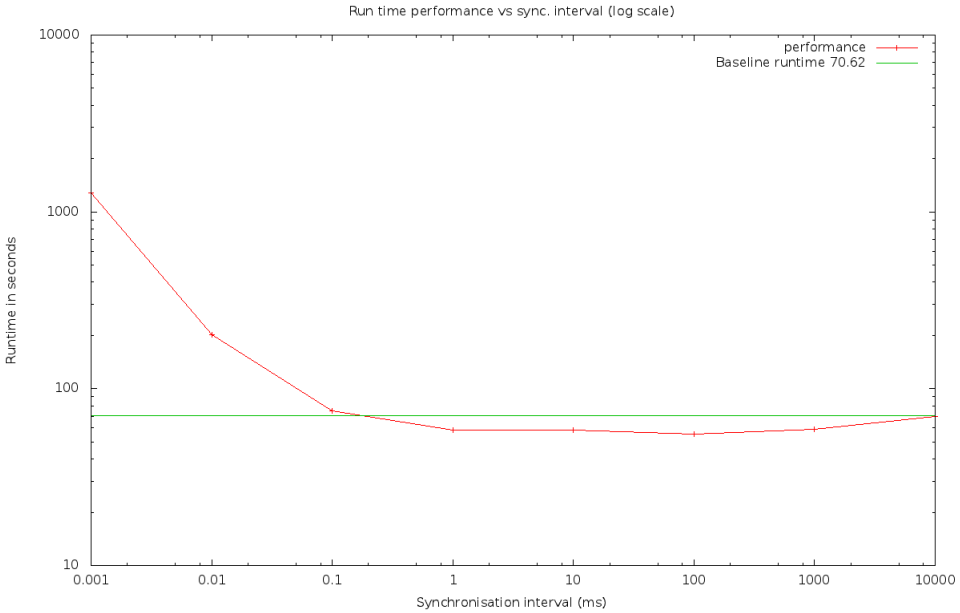

Figure 4-2: Performance of Mobitest

The graphs indicate that 1 ms is a sweet spot for both performance and accuracy for such a system. Beyond this point the performance doesn't improve significantly at higher barrier synchronization intervals and the accuracy doesn't improve significantly at lower intervals. There is no data point on the accuracy graph for the 10 second barrier sync interval, because all packets are lost in the ping test. This is because at higher barrier sync intervals, several pings time out without receiving any response.

4.2 Effect of changing CPU models

One of the benefits of using a full system simulator such as GEM5 is the possibility of configuring more detailed CPU models. We experimented with this by changing the CPU model alone in the ping benchmark tests. We change the CPU model to 'timing' which attributes a different number of CPU cycles to each separate op code. We also experimented with the out-or-order execution model that models an out of order pipeline.

The benchmark we used was the same as in the previous section, one host pinging the other 25 times. We measured the mean and variance of the RTT (accuracy) and the simulation time (performance). Across all 3 models, we saw very little difference in the mean RTT (it was 3.5 ms in all experiments). However, we do observe, that for the same 25 ping benchmark, the atomic(functional) model takes 1 minute and 30 seconds to run. The 'timing' model takes 3 minutes and 47 seconds to run, and the out-of-order model takes 19 minutes and 43 seconds to run. For our target scenarios, we are only interested in higher level metrics such as latency and application throughput and not detailed statistics such as cache misses, memory access latencies and pipeline stalls. In such cases, as seen by the preliminary experiments above, it may be better to use a more simplified and equivalent model than incur the overhead of running a complex model that simulates every microarchitectural detail.

4.3 Simulating Multiple Instances

GEM5 by default, has support only for an Ethernet Link. A switch can be simulated by connecting multiple GEM5 instances in pairs over Ethernet Links and using one such instance as a software router for packets coming from the other instances. However, within ns-3 the system architecture automatically allows us to simulate multiple (≥ 3) instances commmunicating with each other over a wireless broadcast network (simulated by ns-3). This is because ns-3 now takes the responsibility of being the software router. We set up a simple configuration to demonstrate this capability. 3 GEM5 instances were connected over ns-3 using tap devices so that they could ping each other the simulated wireless network. The trace below is after a warm up ping to make sure ARP doesn't skew the measurements.

```
# ping -c 10 10.255.255.255
Pping(685): unaligned trap at 0000000120019024: 000000011ff3bbf4 29 4
Iping(685): unaligned trap at 0000000120019078: 000000011ff3bbec 29 5
NG 10.255.255.255 (10.255.255.255): 56 data bytes
84 bytes from 10.0.0.1: icmp_seq=0 ttl=64 time=0.0 ms
ping(685): unaligned trap at 0000000120019024: 000000011ff3bbf4 29 4
ping(685): unaligned trap at 0000000120019078: 000000011ff3bbec 29 5
84 bytes from 10.0.0.3: icmp_seq=0 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.2: icmp_seq=0 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.0 ms
84 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=1.9 ms (DUP!)
84 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=4.8 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.0 ms
84 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=1.9 ms (DUP!)
84 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=3.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.0 ms
84 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=1.9 ms (DUP!)
84 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.0 ms
84 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=1.9 ms (DUP!)
84 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.0 ms
84 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=1.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.0 ms
84 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.0 ms
84 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=2.9 ms (DUP!)
```

```
84 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=3.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=0.0 ms
84 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=1.9 ms (DUP!)
84 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=2.9 ms (DUP!)
84 bytes from 10.0.0.1: icmp_seq=9 ttl=64 time=0.0 ms

--- 10.255.255.255 ping statistics ---
10 packets transmitted, 10 packets received, 17 duplicates, 0% packet loss
round-trip min/avg/max = 0.0/1.7/4.8 ms
```

Chapter 5

Related Work

MobiTest draws from prior work in several different domains. This chapter presents a high level summary of the most relevant of these.

Network Simulation: Evaluating applications in a virtual environment is not a new idea. Most routing protocols in wired, wireless and ad-hoc networks are evaluated at a large scale using network simulators such as OPNET [15], GlomoSim [37] and ns [6]. Such simulators are heavily geared towards modeling the networking characteristics of the system and go to great lengths to ensure that the MAC and PHY layer conform strictly to published standards. In doing so, however, they ignore the modeling of computation. Simulators typically abstract the computation into one high-level language function representing the entire application of interest. Alternatively, some simulators evaluate only limited functionality. For example: a specific routing protocol devoid of any application context. Abstracting computation achieves scalability but sacrifices accuracy. Furthermore, the application being evaluated needs to be developed twice: once for the purpose of simulation in an abstract form and once for the purpose of actual deployment. This approach is fraught with challenges since it entails additional developer effort. It is also prone to human error during application refactoring. MobiTest aims to allow application binaries to be run unmodified in the real world and in simulation, thereby avoiding this problem.

Compute Emulation in Network Simulations: Emulation [19], on the other hand, gets around the abstraction problem by allowing applications to be evaluated as-is; without any modifications to the original application binary. Several network simulators such as OPNET [15], GlomoSim [37] and ns [6] have support for emulation in addition to their usual simulation mode. However, as pointed out earlier, emulation usually is achieved by synchronising the network simulator's clock to the wall clock time. This approach works until the network simulator can keep up with the packet generation rate from the computer nodes. After a point, the simulator is unable to keep up to real time, skips deadlines and results beyond this point, if obtained at all, are inaccurate. MobiTest with barrier synchonization is a way to sidestep this problem by relaxing the real time constraint and taking a longer time to complete the simulation at the benefit of better accuracy and no real-time deadline missing.

Time Dilation: Systems such as Diecast [21],Timewarp [22] and Slicetime [34] circumvent this deadline missing problem associated with real time systems by exploiting the fact that applications run inside VMs. They use this fact to slow down or dilate time inside the virtual machines as needed by the applications. Such systems trade off simulation time for higher accuracy by relaxing the real time constraint on systems. This also solves the scalability problem since there is no longer the real time requirement to keep up with. However, these systems make extensive use of hardware virtualization, a feature MobiTest cannot assume when the guest and host are different. These also require changes to a native hypervisor such as Xen, which causes portability issues. In contrast, by operating completely as a user space program, MobiTest has far lesser portability concerns.

Full System Simulation: Computer architects use full system simulators such as GEM5 [14] and SIMICS [26] to simulate complete systems running a full blown kernel. The simulated kernel/processors may have no relation to the host kernel/processor. These simulators operate using an event queue with a loop executing events until the queue is empty. The event driven nature of the simulation allows tremendous flexibility in the degree of detail at which the system may be simulated with an associated increase in simulation time. However, these tools are targeted primarily for micro-

architectural design space exploration with the result that simulation of networked systems are not usually possible. Typical use cases of such systems involve single CPU simulations which try and evaluate the impact of microarchitectural changes. MobiTest builds primarily on the GEM5 simulation infrastructure by adding support for network simulation.

Chapter 6

Future Work and Conclusions

This thesis presents several distinct designs for an evaluation infrastructure that can evaluate mobile distributed applications. For each design, we present the attendant accuracy-simulation speed tradeoffs. We also present our experiences implementing each design. Overall, we conclude that the design that uses GEM5 as the simulator for the computation and ns-3 for the network simulation is the most feasible design.

Several aspects of our final design utilising GEM5 and ns-3 can be evaluated in greater depth. First, we have shown how the infrastructure works on an Alpha architecture. Porting this to the ARM architecture would be beneficial to the community of mobile app developers. Further, porting Android, the dominant mobile platform to run atop this simulation infrastructure would also enhance its applicability.

On an independent thread, the support for sensory inputs within GEM5 is limited. On the other hand, mobile phones are characterised by an abundance of sensors. Adding sensor support to a GEM5 supported kernel would widen the range of apps that could be run on such an evaluation infrastructure. The simplest sensor to add would be the GPS, which could be implemented simply by making the simulated kernel read from a GPS trace file on the host machine. Alternatively, the GEM5 console could be modified to accept GPS fixes. This is the approach followed by the stock Android emulator and is used regularly for testing location enabled applications. This console can also be used to simulate other sensory inputs such as the accelerometer, gyrometer, and human key presses at specific locations on the screen.

Such commands would have to be automated in a simulated setting though, since the simulation would run several orders of magnitude slower than real time.

Another completely orthogonal thread of future research is to close the loop between the outputs of the application and the inputs from its users. We touched upon this topic earlier in the Section on human modelling 2.4. This would allow us to take outputs from the applications and simulate their effects on users and simulate user inputs on the applications.

Bibliography

[1] Android developers blog: A faster emulator with better hardware support. http://android-developers.blogspot.com/2012/04/faster-emulator-with-better-hardware.html.

[2] Android emulator. http://developer.android.com/tools/help/emulator.html.

[3] Android kernel features. http://elinux.org/Android_Kernel_Features.

[4] Android-x86 - porting android to x86. http://www.android-x86.org/.

[5] Ieee 802.11p. http://en.wikipedia.org/wiki/IEEE_802.11p.

[6] ns-2. http://www.isi.edu/nsnam/ns/.

[7] ns 3. http://www.nsnam.org/.

[8] Qemu emulator user documentation. http://wiki.qemu.org/download/qemu-doc.html.

[9] Research projects. http://smart.mit.edu/research/future-urban-mobility/research-projects.html.

[10] socat multipurpose relay. http://www.dest-unreach.org/socat/.

[11] Spec benchmarks. http://www.spec.org/benchmarks.html.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[14] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52 –60, july-aug. 2006.

[15] Xinjie Chang. Network simulations with opnet. In *Simulation Conference Proceedings, 1999 Winter*, volume 1, pages 307 –314 vol.1, 1999.

[16] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[17] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[18] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 29–39, New York, NY, USA, 2008. ACM.

[19] K. Fall. Network emulation in the vint/ns simulator. In *Computers and Communications, 1999. Proceedings. IEEE International Symposium on*, pages 244 –250, 1999.

[20] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible mpsoc simulation. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 71–80, New York, NY, USA, 2009. ACM.

[21] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29:4:1–4:48, May 2011.

[22] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[23] Emmanouil Koukoumidis, Dimitrios Lymberopoulos, Karin Strauss, Jie Liu, and Doug Burger. Pocket cloudlets. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 171–184, New York, NY, USA, 2011. ACM.

[24] Emmanouil Koukoumidis, Li-Shiuan Peh, and Margaret Martonosi. Regres: Adaptively maintaining a target density of regional services in opportunistic vehicular networks. In *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications*, PERCOM '11, pages 120–127, Washington, DC, USA, 2011. IEEE Computer Society.

[25] Emmanouil Koukoumidis, Li-Shiuan Peh, and Margaret Rose Martonosi. Signalguru: leveraging mobile phones for collaborative traffic signal schedule advisory. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 127–140, New York, NY, USA, 2011. ACM.

[26] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, feb 2002.

[27] Suhas Mathur, Tong Jin, Nikhil Kasturirangan, Janani Chandrasekaran, Wenzhi Xue, Marco Gruteser, and Wade Trappe. Parknet: drive-by sensing of road-side parking statistics. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 123–136, New York, NY, USA, 2010. ACM.

[28] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[29] Marius Monton, Antoni Portero, Marc Moreno, Borja Martinez, and Jordi Carrabina. Mixed sw/systemc soc emulation framework. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 2338 –2341, june 2007.

[30] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. Wishbone: profile-based partitioning for sensornet applications. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 395–408, Berkeley, CA, USA, 2009. USENIX Association.

[31] M. Piórkowski, M. Raya, A. Lezama Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. Trans: realistic joint traffic and network simulator for vanets. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(1):31–33, January 2008.

[32] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, October 2009.

[33] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. A case for fame: Fpga architecture model execution. In *Pro-*

ceedings of the 37th annual international symposium on Computer architecture, ISCA '10, pages 290–301, New York, NY, USA, 2010. ACM.

[34] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. Slicetime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 19–19, Berkeley, CA, USA, 2011. USENIX Association.

[35] Keith Winstein and Hari Balakrishnan. End-to-End Transmission Control by Modeling Uncertainty about the Network State . In *HotNets-X*, Cambridge, MA, November 2011.

[36] Jungkeun Yoon, Brian Noble, and Mingyan Liu. Surface street traffic estimation. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, MobiSys '07, pages 220–232, New York, NY, USA, 2007. ACM.

[37] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 154 –161, may 1998.