Distributed system simulation with host-based target offloading

BJÖRN RUNÅKER
Distribuerad systemsimulering med värdbaserad avlastning av målsystem

Distributed system simulation with host-based target offloading

Runåker, Björn
runaker@kth.se

Exjobb i datalogi (CSC)

Handledare: Arnborg, Stefan
Examinator: Arnborg, Stefan

Uppdragsgivare: Intel

Datum: 2015-04-06
**Sammanfattning**


**Abstract**

Scaling of TLM (Transaction Level Modeling) simulations for performance is difficult. In this project I will go through several causes of poor performance. This paper describes several simulation engines that use SystemC that are connected so they together simulate next generation radio base station. It also describes how to build a virtual network in, for security reasons, constrained environment where both virtual and physical equipment can connect the simulated target. Furthermore, it describes how to use the virtual network to improve the overall performance of the simulation. It is shown how the virtual network is used to distribute the simulation and offloading the simulated target to accomplish the performance goal.
Acknowledgements

I would like to express my appreciation for the feedback on this report to Gunnar Hallendal and the teams at Ericsson, and especially to the Ghost team, where Ola Dahl, Lei Liang and many other spend time both reading and giving back important input to me. I also want to thank my employer Intel and the simulation team, where Paul Hintikka and many others gave me valuable feedback. Thanks to my manager, Dan Verdin, for allowing me to do this.

I also would like to thank my supervisor and examiner Stefan Arnborg at Royal Institute of Technology.
# Table of content

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nomenclature</td>
<td>1</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Outline of the report</td>
<td>7</td>
</tr>
<tr>
<td>The Problem</td>
<td>9</td>
</tr>
<tr>
<td>Background</td>
<td>11</td>
</tr>
<tr>
<td>Existing research regarding fine-grained simulator parallelism</td>
<td>17</td>
</tr>
<tr>
<td>Implementation of coarse-grained parallelism</td>
<td>18</td>
</tr>
<tr>
<td>Why the current simulator solution could not be used</td>
<td>20</td>
</tr>
<tr>
<td>Alternative solutions considered and rejected</td>
<td>21</td>
</tr>
<tr>
<td>Scapy solution</td>
<td>21</td>
</tr>
<tr>
<td>Raw socket solution</td>
<td>22</td>
</tr>
<tr>
<td>The chosen solution is using a proxy server</td>
<td>23</td>
</tr>
<tr>
<td>Bridging the machines</td>
<td>23</td>
</tr>
<tr>
<td>Connecting test net to test equipment</td>
<td>24</td>
</tr>
<tr>
<td>Anatomy of the Proxy solution</td>
<td>26</td>
</tr>
<tr>
<td>Software implementation</td>
<td>27</td>
</tr>
<tr>
<td>TunnelSetup</td>
<td>27</td>
</tr>
<tr>
<td>Handling exceptions</td>
<td>28</td>
</tr>
<tr>
<td>TAPmanager</td>
<td>29</td>
</tr>
<tr>
<td>Tapdaemon</td>
<td>29</td>
</tr>
<tr>
<td>Portsetup</td>
<td>30</td>
</tr>
<tr>
<td>Extee</td>
<td>30</td>
</tr>
<tr>
<td>setIPfromMac</td>
<td>31</td>
</tr>
<tr>
<td>Tssocks</td>
<td>31</td>
</tr>
<tr>
<td>TunnelingRecursiveRouter</td>
<td>32</td>
</tr>
<tr>
<td>The proxy solution version 2 (TRR) diagram</td>
<td>33</td>
</tr>
<tr>
<td>Connecting two different virtual nets using TRR</td>
<td>34</td>
</tr>
<tr>
<td>Increase performance by moving execution out of target</td>
<td>39</td>
</tr>
<tr>
<td>Conclusions</td>
<td>41</td>
</tr>
<tr>
<td>Future</td>
<td>41</td>
</tr>
<tr>
<td>References</td>
<td>42</td>
</tr>
</tbody>
</table>
## Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADK</td>
<td>Axxia® Development Kit</td>
</tr>
<tr>
<td>ARM®</td>
<td>Processors based on a Reduced Instruction Set Computing (RISC) architecture developed by the British company ARM Holdings. (Wikipedia, 2015a)</td>
</tr>
<tr>
<td>ASE</td>
<td>Axxia® Software Environment</td>
</tr>
<tr>
<td>AT</td>
<td>Approximately timed – (context: TLM) Every model synchronized to a common simulation time. Each process runs at a specific simulation time, according to when the activity happens in the target system.</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit – in this system it is where, for example, the DSPs are located.</td>
</tr>
<tr>
<td>Axxia®</td>
<td>Intel® processor architecture for telecom</td>
</tr>
<tr>
<td>CoSim</td>
<td>Used in DVP to connect simulation of AXM processor with SVP. See EMA.</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic Host Configuration Protocol</td>
</tr>
<tr>
<td>DPI</td>
<td>Deep Packet Inspection – Using regular expression to find pattern within a data packet.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DTB</td>
<td>Device Tree Blob - The Device Tree is a data structure for describing hardware. Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time. (&quot;Device Tree,&quot; 2015). See also: “Device Tree for Dummies” (Petazzoni, 2013)</td>
</tr>
<tr>
<td>DVP</td>
<td>Digital unit Virtual Platform – system that consists of several simulator engines to create major part of a base station. The main components are ASE and SVP.</td>
</tr>
<tr>
<td>EMA</td>
<td>External Model Adaptor – Proprietary library for connecting SystemC based simulation engines using TCP/IP. In this paper CoSim has the same meaning as EMA, but Ericsson has a different definition for CoSim, which is outside the scope of this paper.</td>
</tr>
<tr>
<td>Erlang</td>
<td>Programming language (Ericsson, 2015c)</td>
</tr>
<tr>
<td>Erlang OTP</td>
<td>Open Telecom Platform library for Erlang (Ericsson, 2015e)</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array – Unlike ASIC, logic gates are programmed by software when system is started and not at manufacturing.</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile communication</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-Integrated Circuit - I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. (Wikipedia, 2015c)</td>
</tr>
<tr>
<td>IP Block</td>
<td>Semiconductor intellectual property. Functional blocks that can be licensed from the owner of the design so it can be included in a chip (Wikipedia, 2015m)</td>
</tr>
<tr>
<td><strong>IPsec</strong></td>
<td>Internet Protocol Security - (Wikipedia, 2015e)</td>
</tr>
<tr>
<td><strong>L2</strong></td>
<td>Layer 2 Data link layer (e.g. network switch) - (Wikipedia, 2015i)</td>
</tr>
<tr>
<td><strong>LT</strong></td>
<td>Loosely Timed - (context: TLM) allow models to run ahead in their own time within a time limit (time quantum) before waiting for other processes that need to catch up. Simulate as-fast-as-possible.</td>
</tr>
<tr>
<td><strong>LTE</strong></td>
<td>Long-Term Evolution, commonly marketed as 4G LTE, is a standard for wireless communication of high-speed data for mobile phones and data terminals. The standard is developed by the 3GPP (3rd Generation Partnership Project) and is specified in its Release 8 document series, with minor enhancements described in Release 9. (Wikipedia, 2015g)</td>
</tr>
<tr>
<td><strong>NTP</strong></td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td><strong>OSCI</strong></td>
<td>Open SystemC Initiative - The Open SystemC Initiative (OSCI), a collaborative effort to support and advance SystemC as a de facto standard for system-level design. SystemC is an interoperable, C++ SoC (System-on-a-Chip) / IP Block modeling platform for fast system-level design and verification. Now merged with Accellera (Accellera, 2015a)</td>
</tr>
<tr>
<td><strong>Proxy</strong></td>
<td>Used in this context as a short for proxy server, a computer network service that allows clients to make indirect network connections to other network services.</td>
</tr>
<tr>
<td><strong>QoS</strong></td>
<td>Quality of service – quantity of specific network performance such as bandwidth, throughput, jitter, latency, etc</td>
</tr>
<tr>
<td><strong>RAT</strong></td>
<td>Radio Access Technology (e.g. GSM, UMTS or LTE)</td>
</tr>
<tr>
<td><strong>RTL</strong></td>
<td>Register-Transfer Level is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. (Wikipedia, 2015k)</td>
</tr>
<tr>
<td><strong>Scapy</strong></td>
<td>Python library for packet decoding and creation. (Biondi, 2007)</td>
</tr>
<tr>
<td><strong>SSH</strong></td>
<td>Secure Shell (Ylonen) – Establish a secure channel over an insecure network. In this project it is used to connect sockets on different hosts.</td>
</tr>
<tr>
<td><strong>SVP</strong></td>
<td>System Virtual Platform – Simulation of ASIC containing DSPs. Part of DVP.</td>
</tr>
<tr>
<td><strong>SystemC</strong></td>
<td>SystemC is a set of C++ classes and macros which provide an event-driven simulation interface. (Wikipedia, 2015o) Documentation found at (Accellera, 2015c)</td>
</tr>
<tr>
<td><strong>TAP interface</strong></td>
<td>A network tap that simulates a link layer device and operates with layer 2 packets like Ethernet frames. (Wikipedia, 2015q)</td>
</tr>
<tr>
<td><strong>Telecom</strong></td>
<td>Telecommunication. Used here to describe mobile telephone network base station equipment.</td>
</tr>
<tr>
<td><strong>TLM</strong></td>
<td>Transaction-level modeling is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. (Wikipedia, 2015s) Higher level of abstraction compared to RTL.</td>
</tr>
<tr>
<td><strong>UMTS</strong></td>
<td>Universal Mobile Telecommunications System (3G)</td>
</tr>
<tr>
<td><strong>Virtio</strong></td>
<td>(Context: Linux) API optimized for virtual I/O. (RedHat, 2015)</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network. Most modern WLANs are based on IEEE 802.11 standards and marketed under the Wi-Fi brand name. (Wikipedia, 2015u)</td>
</tr>
</tbody>
</table>
Introduction

Telecommunication technology is today clearly visible in the public sphere: in transportation vehicles and on the streets, where a large portion of the people are visibly engaged in checking up their social media news, watching cats on YouTube, muster prospective dates on dating sites, reading eBooks, shopping with search engines, and starting the day’s work on their connected laptops when they enter the suburban shuttle vehicles.

In suburban environments, more and more people use streamed music and video. More and more of the user's information are stored in the cloud, updated from anywhere in collaborative writing and downloaded repeatedly. Passengers on cruise ships and airplanes expect the same accessibility as when they are at home or in office, so they can interact with friends and neighbors, or perform business.

Behind these visible scenes there is large machinery devoted to receiving and delivering an information traffic that increases continuously and rapidly. New technologies in fiber, microwave and satellite communication, guided by advances in optical transmission and antenna technology, increase the bandwidth and improve reach of links.

The present work is connected to the sorting and delivery part of the Internet: routers that receive packets, cache on disk, read addresses, encrypt and decrypt, and decide best routing paths. The computing parts of routers were originally standard general-purpose computers, but today they are often implemented with very specialized hardware to handle high volume traffic. Figure 1 shows an estimate of the future growth of data that are transferred over the Internet.

![Figure 1: Projected Internet traffic per month (C. S. v. Statista, 2015)](image)
The telecom equipment used in this work is not only handling the standard Internet routing, it also connects mobile devices using several telecom standards such as 2G (GSM), 3G (UMTS), 4G (LTE) and WLAN. The processor used in this study is typically used in telecom equipment such as base stations. The role of the base station has changed from mainly connecting voice calls to route data traffic to mobile devices. As seen in a report from Cisco VNI, Figure 2, the increase of data traffic on mobile devices is expected to grow tenfold, which will require a massive upgrade of current installation of base stations.

![Global Smartphone Traffic to Increase Tenfold by 2019](image)

*Figure 2: Projected mobile traffic (C. V. v. Statista, 2015)*
Ericsson did a comparison of the actual amount of mobile data to voice during the last four years, Figure 3. Here, it is obvious how base stations do more and more IP routing, and the traditional voice calls get a smaller percentage of the total traffic.

As the growth rate of data traffic increases, the time to develop new generation base station must be even more quickly executed, which is further complicated by the fact the traditional solution does not fit the future use case of base stations.

The rate that the market demands a better performance, smaller size while using less energy, is increasing all the time. This means if a company wants to stay competitive today, the lead times from decision to deployment must be shortened. This also means that software development cannot wait for the hardware to be available to start testing the software implementation in order to reasonably predict performance and check correctness.

Using simulators, where software can be tested while the hardware is still under development, solves the problem with developing software for a system not yet available. As long as the simulator has enough performance, so developers find it feasible, as a tool and give the result within reasonable time. How this performance can be improved for software developers will be described in this report.
Outline of the report

The current work describes a delimited project in simulation systems for telecom processors. In this report, I will show how we improved performance of the simulation using host-based target offloading. How this was accomplished and what kind of problems we needed to solve will be described.

NOTE: What kind of processors Ericsson actually will use in future development is not discussed in this paper. Test systems are developed all the time to investigate the feasibility of different architectures.

During my work as a support engineer where I help software developers at Ericsson to program our AXXIA® network processor, I was confronted with a more difficult problem that at that time was seen as a blocker to make a simulation of an almost complete radio base station. The problem will be described in the chapter “
Why the current simulator solution could not be used. I then present my solution, which will later be called “Proxy solution”. This solution was at first considered quite complex, and we had to show that there was no other simpler solution. The alternative solutions we found during the investigation phase are listed in the chapter “Alternative solutions considered and rejected”. There it is described why they do not fulfill the necessary requirements. We were running late in the project, and no other solutions were available at the time and some people even deemed the whole thing to be unsolvable! I was given the opportunity to implement my proposed solution. Ericsson gave me access to a couple of “sandbox” servers where I had full root access. I also had direct contact with the team (based in Gothenburg and Ottawa, Canada), which was the first user of the Proxy solution, in that way it could be sure that it was implemented in such way so it could be used directly. Being a one-man team (with all the necessary support from Ericsson) made the project management very easy, compared to how it was usually done. After every release of my software, I got response within couple of hours so the turnaround times were quick. I also did the integration of my solution into specific scripts used at Ericsson together with the teams at Ericsson, but these scripts are not presented in this report because they contain proprietary code and should not be interesting for anyone outside Ericsson.

To summarize, this report will describe an overview of the simulation used in this project, moreover describe some performance issues we struggled with, and how they were solved.
The Problem

This report will first describe the performance problems with the TLM simulation system jointly developed at LSI and Ericsson, and then describe the network problem that was a showstopper until it was solved. It is theoretically impossible (or at least very difficult) to increase the simulation speed in the same way as the chip designs improve performance. The reason is that more and more of the improvement comes from parallelism inside the chip and current simulation techniques are not as good on running in parallel. This means that each generation of chips that improve according to Moore’s law, will not see comparable improvement on the simulation speed of the same chip.

One of the performance problems we could solve was that the disk access from the simulation was very slow. This was solved by using something called virtio (RedHat, 2015) that was introduced to Linux just to improve disk performance on Linux running inside a simulation. Although it improved the performance, we found that the native disk performance has a large effect on the overall simulation performance. We measured how the boot time for Linux on target changed depending on where the native file system resided. We did several measurements of performance and found that the native disk performance not only had an effect on the simulated disk performance, it also had an effect on the performance of other components. In Table 1 it is shown how the performance of simulation of flash is affected by disk performance as can be seen in the first column, which represents the improvement when we had a simulated flash that was accessed during the boot phase. Internally in the simulator the flash used a disk file, so this explains the effect of the native disk performance. The second column shows less improvement because less disk access was performed during boot up. The percentage how much the execution time is improved compared running on NFS mounted disk system to a local RAM disk.

<table>
<thead>
<tr>
<th>Linux boot time</th>
<th>Flash present and accessed</th>
<th>No flash present</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance improvement when changing from NFS to RAM disk</td>
<td>45.5%</td>
<td>24.1%</td>
</tr>
</tbody>
</table>

*Table 1: Improvement of simulation performance depending on file system location*

Other problems were the accelerators that were available on the target and were not simulated efficiently. The most important accelerator was the cryptography engine, which resulted in very slow simulation of key generation. This is something that is done during the first boot and these keys are part of certificates that will later be used by the target to set up secure communication.

Next we found, when we attached external network connection to the simulated system, what would become our main problem, and it was related to the security policy of the enterprise network. There were several problems with the proposed solution that the security department objected against. First, having an unmanaged Unix computer with
unauthenticated users with root access connected to company network is totally prohibited. Secondly, doing modification as a super user to the high performance servers used to run the simulation is not allowed, the reason for this is both for security and for the way the servers are managed by the IT department.

The first part of this report will describe how the simulated target gets network access, which fulfills the constraints that the security and IT department impose on the system. The second part will focus on the virtual network solution and explain what it is. I also describe the reason behind the chosen solution, where security concerns, and several other constraints demanded a solution like this. Then, we show how to use this network capability to offload the simulation so much more performance can be achieved and still fulfill the important requirement with exactly the same binary software image being used on both the simulated target and the physical target.
Background

The Axxia division of LSI Corporation (now part of Intel) is producing one of the most complex networking processors in the world that consist of a multitude of different IP blocks that solve and accelerate different functions that are needed in high-speed network appliances used in telecom systems.

This is how our public press release describes the chip: “The Axxia® Communication Processor AXM5500 product family is designed to accelerate the performance, power efficiency and integration demands of next-generation mobile networking equipment.” ("Axxia® Communication Processor AXM5500 Family," 2014).

What we describe in this paper is about to simulate the next generation of AXM, which contains several fundamental improvements, but most of the ideas behind the architecture are similar. This chip has no public press release nor available to public yet, so I describe the architecture of AXM5500 instead. The press release of the AXM5500 describes very well what makes AXM5500 processors so special: “The AXM5500 product family achieves its performance efficiency by providing a flexible combination of general-purpose processors and specialized packet-processing acceleration engines. These processors and engines use Virtual Pipeline technology, a message-passing control path, to efficiently and autonomously process packets. This approach provides system designers with the ability to define packet-processing flows using multiple combinations of acceleration engines and CPUs to meet a wide range of application requirements. In mobile network applications wireless transport processing can be implemented in the Virtual Pipeline and acceleration engines, allowing for greater performance and power efficiencies.” ("Axxia® Communication Processor AXM5500 Family," 2014)

The architecture diagram below, Figure 4, shows how a cluster of general-purpose CPU cores is connected with a “Virtual Pipeline Task Ring” to numerous acceleration engines. This is used to move data quickly between different processing units inside AXM5500. It is called “virtual pipeline”, which is used to define different paths that each data packet will follow from ingress (typically entering the chip through an Ethernet interface) to egress (typically leaving the chip on a different Ethernet interface). There are several processing units where a decision can be made, based on different classifications of the packet or based on stored data associated with the packet’s data stream, which will choose the virtual pipeline that the packet will continue on. The virtual pipeline is defined when the system is configured, but each data stream that is flowing through the system will have dynamic data during the lifetime of the data stream. Examples of this data, that is also called flow data, are QoS (Quality of Service) of the packets in the data stream, and storage of packet fragments during reassembly.

The CPU clusters are inter-connected using ARM® CoreLinkTM CCN-504 Cache Coherent Network that is QoS-aware and optimized for networking workloads ("CoreLink CCN-504 Cache Coherent Network," 2014). The important accelerator engines are the packet processor (handle up to 50Gb/s), the security protocol processor (handle up 25Gb/s), StreamSight™ DPI Engine (handle up to 7Gb/s), traffic manager (7
levels of hierarchy) and High-Capacity Ethernet Switching doing non-blocking layer 2 switching (up to 160Gb/s).

![AXM5500 Architecture Diagram](image)

**Figure 4:** AXM5500 Architecture Diagram

It is the complexity of the processor that makes it necessary to have tools like a simulator to be able to understand and debug the software that is being developed on it. For that reason we developed the ASE tool suite, an Eclipse (Eclipse Foundation, 2015) based development environment, which contains a configuration tool for creating “virtual pipelines” and a functionality to build binary firmware for the accelerators and to simulate the whole chip.

Accelerators are specialized hardware implementation of functions that are typically implemented in software using general-purpose processors. Examples of accelerators are security protocol processing, checksum calculation, reassembly, packet classification, etc. The virtual pipelines are the connections that define the accelerators in a specific order to handle a classified packet from a data stream. For example, there are virtual pipelines that do routing, handles traffic policing (monitoring network traffic for compliance with traffic contract, and taking steps to enforce that contract), shaping (delay some network traffic to bring them into compliance with a desired traffic profile or QoS), IPSec, etc.

Using the ASE environment, we can build test cases with several virtual pipelines, with configurations for each accelerator (a combination of static parameters, a C like language and a rule matching language) and a virtual traffic generator / analyzer. When we run the tests, we will see if the data sent into the simulator generates the correct output according to specification. We also see for each use case how many resources
were used from each accelerator and the bandwidth (data transfer rate) we achieved. There are numerous counters in the chip that shows how much of the queues were used (the high-water mark), memory usage, dropped packets, etc. From this we can both find faults in the logic of the code and find where the cause of a below average performance resides and how to optimize that part. Because of this we used ASE as a teaching tool for the software teams so they could understand how to use the accelerators.

What I have found during my time as Field Application Engineer is that a common problem is moving an existing software design that uses a generic CPU centric solution to use as much as possible of the accelerator engines that solve a specific task much more efficient. One of the reasons we developed the ADK (Axxia Development Kit) is to simplify the usage of the accelerator engines. The ADK contains premade virtual pipelines and programs for the accelerator engines that implements different layers of network protocols. These modules can then be connected together to implement a complete protocol stack. For example there is one ADK module that implement Ethernet functionality (i.e. mac address handling, checksum calculation, etc.) and another module implementing IP and UDP. These can be connected together to get full UDP/IP functionality without writing any accelerator engine code.

Many standard Internet protocols and most telecom protocols are available in ADK. Propriety protocols can also be implemented alongside with the ones provided by ADK. The ADK is mainly used in the data path, but the simulator described in this paper will focus more on the control path applications, so the ADK will not be described in depth in this paper. Note that in telecom the concepts of control path and data path have a specific meaning. The control path is used to setup a session (e.g. connect a caller with a recipient) and other management function, and the data path generally contains the actual data that flows through the system from end-to-end (e.g. data traffic, voice, etc.).

The main data points that are used when benchmarking a solution are how many simultaneous sessions (data streams) and the latency and bandwidth on each session that this part of the system can handle. That is why it is important to run each use case through the simulator when selecting different ways to refactor the code to use different sets of accelerators. In each case there could be different bottlenecks that limits the performance. Note that this chip has multiple 10G interfaces, so the type of processing needed for each data stream will mean that the packet rate is too high to be handled by a generic CPU. Each packet will typically be classified, shaped, scheduled, reassembled, security protocol processed, etc. Specialized accelerator engines that are available in the Axxia processor should handle all this and more. But the most common design fault is to route a data path up to the CPU cluster, when it could be instead solved using the network accelerators. The main problems with including CPUs in packet processing in the data path are that the generic CPUs get very easily saturated, and do not have deterministic enough response time (especially when using Linux, even with real-time patches) and low I/O bandwidth compared with the Axxia accelerators. It is a different story when considering the control path, where the general purpose CPUs is the appropriate component to handle this. The control path is where the system setup is done, and it is also an appropriate place to handle exceptions (e.g. fault situations) and other uncommon situations. This usually needs a more complex software design that is best handled by general purpose CPUs.
The need of using a simulator for software development is increasing with the requirement to decrease the time to market. The common practice has been to develop software on available hardware, which means that some part of the software testing will be delayed until the hardware is available. This is unacceptable where time to market with new products is important. Because so much of the product development time consists of software engineering, the impact on the release schedule will be very large. This makes it necessary to start the software development cycle while the hardware (including the silicon) is not available.

What makes simulation even more important today is that the interaction between CPUs, acceleration engines and DSPs are increasingly complex for each generation of chip architecture. This increases the software engineer’s need for tools that help them to understand the system and where they can easily test modules so it is easy to investigate the software’s behavior. The system that is simulated consists of a generic CPU part (a cluster of ARM cores) that is quite easy to simulate, but the complex parts to simulate are on chip network accelerators, an external ASIC with several DSPs, FPGA and the interaction between these. Part of this specific system’s ASIC part is described in Design and Implementation of an Extendable SoC Virtual Platform in SystemC-TLM 2.0 (Liang, 2012).

The simulator needs to be tuned for software development, which means it must be fast enough to be used in debugging high-level software logic. Low-level simulators (for example on RTL level) are too slow to be useful for high-level software developers. Just booting Linux would in most cases take hours.

We are using a TLM based simulation of each component in the system. The difference between a RTL based simulations is cycle accurate with the real hardware (actually TLM 1.0 can also do simulation that is cycle accurate). But TLM 2.0 is either AT (Approximately-Timed) or LT (Loosely Timed), and because we simulate the system for the sake of the programmers, our focus is on LT, as highlighted in Figure 5.
In LT a discrepancy is allowed between exact time for each operation and simulated time. This allows optimizing the execution performance by reducing the amount of synchronization between events.

Each simulated component is written using SystemC C++ classes. As shown later, this is a higher level of abstraction where some details of the actual implementation of HW are hidden, so the performance of running the simulation is improved. It also improves the simulation of data transfer, where, for example, I^2C data transfer is represented as a function call with a parameter. The parameter is a buffer containing all the bytes that are transferred in an I^2C message. But an actual I^2C transfer consists of a data signal line that goes high or low in synchronization with a clock signal. Simulating each step of these bit-wise transfer steps in a simulator would be very time consuming and also uninteresting for a software developer. The whole transfer is abstracted so all the data bit and clock cycles that has been done for a transfer of a message from one device to another on a specific I^2C bus are represented by a buffer with the complete message. There is also some meta-data containing timestamp so the simulator engine can schedule when the I^2C data transfer is actually done (and also how long it takes so it knows when the bus is available for other transfers).

There are numerous corners that are cut to improve the simulation performance, and there are parts that are not even modeled because of they are either too time consuming to simulate or to model or it is unimportant for the overall simulation. Example is the memory bit error registers that the memory controller uses to indicate faulty memory chip. In the specific model used here we never indicate errors in these registers. It is enough that they indicate the non-error value necessary so the application doesn’t halt the execution.

Even with all the performance enhancing simplifications of the models there are limitations when scaling up the simulation. The simulation engine for the ARM model is implemented to use a single process for executing all ARM core models. This means that the real execution time will be longer when you add cores. The parallelism that exists within the simulation in the form of a cluster of 16 CPUs is serialized in the simulation engine.

In the current simulator we have partitioned the system simulation between two simulator engines connected using a CoSim interface (also known as External Model Adaptor, EMA). The choice of how to split the system between the simulators was decided by the chips involved. The processor clusters with the accelerators are simulated in an engine called ASE and the ASIC with DSPs are simulated in an engine called SVP. The main communication between the chips is done via a number of 10G Ethernet connections. This means that the simulations are loosely coupled and a preferred partitioning between the engines. Although each simulator engine is running in its own process and is sequential in execution, in relation to each other they are totally asynchronous with all synchronization done in the connecting adaptor. Although both simulation engines are using LT simulation, the time discrepancy between the simulations can never be larger than a predefined time quantum.

Moving these two simulation engines to different hosts with only network connection in between would reduce the performance considerably because of the bandwidth.
requirement between these two simulations, which actually represents the chips on a real board. This in turns translates to simulator events that are communicated over the adapter in a rate proportional to the real hardware bandwidth requirements. This will be the bottleneck for the total simulation performance unless the bandwidth requirements for the adapter are fulfilled. In a future design of the real system we might share memory between CPU and DSP complex so they loose the quality of being loosely coupled and it will in turn make the current partitioning less feasible. The main performance constraint will then be the use of different processes (and memory areas) for the two simulator engines. There are a number of possible solutions, and one is to merge the simulator engines into one so they run in the same process context, and another is to create a shared memory between the processor engines. There will still be synchronization problems to be solved, but the main bulk of data transfer would be a magnitude quicker than transferring over a network socket. But probably the bridge will still be implemented as sockets and rely on the loosely coupled nature between the simulator engines. The reason for this solution is that SystemC 2.0 does not have the necessary synchronization mechanisms when using shared memory. This is one area we are currently investigating how to solve. For example as seen in Figure 6, if we add a hand-over message of the ownership of the structure in shared memory for the message payload, it would solve the memory access synchronization between the processes (i.e. simulation engines). Note that all simulation engines needs to reside on same host and not be connected over a network to benefit execution speed from shared memory.

Figure 6: Proposed CoSim improvement
But the situation is quite different for other parts of the system, which is outside the simulation engine. In this system, see Figure 7, there are services that are connected over network in the real system and these parts can also be located on different systems in the simulator. Here we can transfer data in the same way as in the real system and simulator specific protocol is not needed.

![Figure 7](image)

Here we get a natural distribution of the workload where services and test applications are residing on different servers (if needed), and an easy way to scale the system when we need more performance.

**Existing research regarding fine-grained simulator parallelism**

Although there are a number of ways to create fine grained parallelism into the simulator, as for example demonstrated in *Improving parallel MPSoC simulation performance by exploiting dynamic routing delay prediction* (Roth, Bucher, Reder, Sander, & Becker, 2013), where it states “Unfortunately, the standard SystemC simulation kernel distributed by the OSCI consortium is sequential, and cannot exploit the parallelism of multi-cores workstations.”. The core idea is interesting how they use dynamic time quantum and still using cycle accurate TLM! But running it required a special SystemC simulation framework (Roth et al., 2012).
An efficient hybrid synchronization technique for scalable multi-core instruction set simulations (Bo-Han, Ren-Song, & Ting-Chi, 2013), where they propose a way to parallelize the simulation by always executing the simulation on oldest time in a synchronization point. The hybrid part is that it uses polling when requesting current simulation time on other hosts, but using collaborative yield when synchronizing on the same host. Utilizing this technique would require major rewrite of the simulator engine.

An Ultrasyncronization Checking Method With Trace-Driven Simulation for Fast and Accurate MPSoC Virtual Platform Simulation (Yu-Fu, Hsin-Cheng, & Chung-Yang, 2013), where they investigate a method to reduce the number of synchronization points. There is an interesting method of using trace-driven simulation. But it seems to require a lot of extra effort to record all the traces needed to identify where it is negligible data dependency so synchronization could be skipped. Anyway it is not a change we could easily do to our simulator engine.

Parallel programming with SystemC for loosely timed models: A non-intrusive approach (M. Moy, 2013), where the addition of concept of duration makes it possible to run in parallel. This seemed to have the best fit for this project, because it used an unmodified SystemC kernel and utilized loosely timed system. This uses sc-during (Matthieu Moy, 2015) that runs on SystemC and creates host threads from SystemC threads. This is perfect to increase the performance of loosely timed systems, but it requires that the models needs to be rewritten to utilize this library. Although it is not possible to rewrite all models in this stage of the project, we are investigating the feasibility of an iterative process where we rewrite a few of the models that would benefit the most of parallelism.

Although fine-grained parallelism would be beneficial for performance, it is not feasible in this case, beside of implementing "sc-during" on a few models. The simulated system in this study consists of several different simulation engines, and improving one of them by using fine-grained parallelism will not increase the overall performance because the simulation time can't progress until the slowest sub-system is done with its time quantum of simulation. Because, in this case, the simulated system includes simulator engines that are not fine-grained parallelizable, like the ARM's FastModel (the implementation is an opaque block which we can't easily modify) of the CPU cluster, fine-grained parallelism of the whole system will not be possible.

Implementation of coarse-grained parallelism

It is important to understand what SystemC is, to understand our approach to parallelism. The SystemC Language Reference defines SystemC as “The SystemC library of classes and simulation kernel extend C++ to enable the modeling of systems. The extensions include providing for concurrent behavior, a notion of time sequenced operations, data types for describing hardware, structure hierarchy and simulation support.” (Accellera, 2015c)

We use a CoSim (or External Model Adaptor) approach that have some similarities to A Cosimulation Framework for a Distributed System of Systems (Muller-Rathgeber &
Rauchfuss, 2008), but in our case we have two different simulation engines that both use SystemC. This has the benefit of the low-level SystemC events can be sent between the systems. The communication between the systems is transferred over a CoSim adapter. It is a software module that is shared between the simulation engines, and contains the protocol and interface. It is through the CoSim adapter that events between the simulations are transferred with correct simulation time. The two simulator engines represent roughly different chips on the circuit board. One handles the network processor, which contains the CPU cluster and network accelerators, and the other an ASIC with several DSPs, a FPGA, and other network accelerators. The CoSim adapter represents the data buses that goes between these simulator engines, and each bus get its own data channel through the adapter. The main data channels are: Ethernet, I²C, SPI and PCIe. This division of simulator engines do actually give the complete system a certain level of parallelism. Each engine will use all available process time on one core each. This is possible because the ASE and SVP runs in different processes and are loosely coupled, which means that one simulator is not idling while waiting for events from the other simulator (as long as they both are on the same time quantum). But both are heavy loaded with jobs and utilize all available computational resources to execute the simulation.

As seen in Figure 8, a typical simulation run uses 2 CPU cores that are running at 100% with a third CPU core used less. **But this parallelism is not enough to get the required performance.**
Why the current simulator solution could not be used

As shown previously it is difficult to scale the simulation using parallelism. This means that the viable solution is to use high performance servers that even without using massive parallelism achieve quick execution time. This also means the servers are expensive and not used more than necessary. The IT department that grant us access to the very high-end servers enforce a strict policy that no configuration changes can be made to the operating system on these systems, because it would increase the downtime between different usages of these servers.

This limitation blocked a vital part in the system, which is the TAP interface that is needed to connect network interface in the simulation to the network stack on the host server. The problem is that we have a socket from the simulator engine that represents an Ethernet interface inside the simulation. This socket is supposed to be connected to a TAP interface on the host running the simulator. Only someone with root privilege can create TAP interfaces in Linux operating system, and not only is it not allowed for us to have root privileges on these machines, we were not even allowed to have any TAP interfaces on these big servers. This is the problem briefly. At this time, the project of simulating a complete base station was almost canceled because the simulated system was not useful without a network connection.
Alternative solutions considered and rejected

We investigated different ways to connect to the sockets that the simulation engines provided as a raw Ethernet packet interface into to the simulated Ethernet interface. At first the problem seemed simple to solve using, for example, Python and a library for building arbitrary Ethernet frame, for example Scapy (Biondi, 2007), pymac, dpkt etc. This would solve the problem with creating packets that are correct for the Linux network stack inside the simulator. But the test tools outside the simulated environment, for example traffic generator, expect to connect to an IP address, not a socket. This means that every test application needs to be rewritten to use the interface that can be provided using this type of scripting.

Scapy solution

Scapy is a versatile tool for all kind of network investigation. A good description of Scapy can be found on its homepage: **Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more.** (Biondi, 2007).

It is quite easy to implement a protocol in Scapy so it responds correctly according to any network protocol (see Figure 9). Even part of SSH can be implemented (tintin, 2014), but much more work is required if you don’t just want to terminate the data stream within Scapy. You have still not attached to the Linux network stack, so any test application that you want to connect via a Scapy solution need to use an interface that can be created in user space, and that is not to create a real network device. Most applications used in this context expect a network interface that you connect using TCP/IP, and not a socket port. Although it is possible in theory to work around this, for example create a socket server in Python and implement all used protocols using Scapy and Python, there are number of problems with such a solution that makes it unfeasible. First implementing every network protocol correctly in Python is a huge effort, and not only would it be expensive compared to the network stack given for free in Linux, it would also be very slow.

![Figure 9: Scapy framework](image)
Raw socket solution

Another solution we considered was to write a special network server application within the simulation framework that connects directly to the network interface and sends high layer OSI model protocol data (for example layer 4, TCP socket level, or even layer 7 so it can attach directly to a test application on host side) as seen in Figure 10. This would remove the need to handle raw Ethernet frames on the host side, but it would basically limit the number of simultaneous protocols to one at the time.

![Diagram of socket connection](image)

Figure 10: Creating a socket to socket connection

**Note:** In the above solution we use the simulator engine’s network socket to directly connect to a test application. The network stack will be totally by-passed on the inside of the simulated Linux and a RAW socket connection is done directly to the recipient application inside the simulator.
The chosen solution is using a proxy server

We got approval to use one machine that could be configured in almost any way I wanted. It did not have enough performance to run simulation engine, and still not allowed to run any application with root privileges, but I could get it configured with TAP interfaces. The TAP interfaces have no direct access to the physical network. This would be our end-point connected to remote machines with simulator engines. We got a class C IP address space that was allocated only for our use. To utilize it to the maximum I designed the network number bit field to be 30 bits, this created a number of networks where each network had only 2 hosts (the two other bit combination was used for broadcast and network identifier). It left me with 6 bits to use as network number, so it allowed me to create 64 separate virtual nets.

![Figure 11]

All the TAP interfaces were placed on a local network (127.0.0.2) so it was not accessible from the external network adapter. Also by putting each TAP interface on a separate network number, the TAP interfaces could not connect to each other without explicitly allow it via a router component. Theoretically, if the network interfaces were put into promiscuous mode it would be able to sniff traffic on the other virtual networks, but was still limited to only allow the traffic on the local network (127.0.0.2).

Bridging the machines

Now we need to connect the sockets that are residing on separate machines. The first version of the solution used SSH tunnels. The reasons we used SSH tunnels are several, first all servers have SSH server installed by default, and second reason is the ability to create tunnels between sockets on different hosts, and the third reason is security. To create tunnels there needs to be a SSH session setup between the server with the simulator instance and the proxy machine.

Because there are other constraints that require the creation and removal of the tunnels dynamically and without changing the SSH session, I used a feature added to SSH 2005 (added in OpenSSH version 4) called multiplexing. The principle is that a master connection is made using SSH and a socket, called control master is created. This socket is used when creating or removing tunnels. The time to setup a new tunnel will be quick because it uses the already established SSH session without setting up a new session key.
etc. The tunnels will be “multiplexed” into the SSH connection and added to any other tunnel already setup in the session.

So what is a tunnel?
A tunnel consists of a listening (server) socket in one end and a connecting (client) socket in the other end. The traffic in the tunnel is bidirectional. In Figure 12, you see a tunnel is setup by a SSH client to a SSH server, but note that the SSH client has started a listening socket server, where applications C1, C2 and C3 have connected to. On the SSH server side, it will create a socket client for each connection the applications does and connect to the service S. It means there will in this case be three connections from the SSH server to S.

![Figure 12: SSH tunnel](image)

**Connecting test net to test equipment**

A more complicated case is how to connect the test equipment to the simulated system. The equipment range from services needed in the “back-bone” that is expected by base-station to run. Example of services are: DHCP, NTP, file server, etc. There is also a need to generate use cases of traffic. And very important is to connect test framework for unit testing of functions. It consists of a test of a specific function where the result of the execution of the function is checked against the expected result. If the expected result is not returned from the function, the test fails. Another very important requirement is that the test framework should be the same when running against a simulated system and when running against real hardware. There are several reasons for this; one is that the cost of the test suite would increase if it has to be a special version for the simulator. Another reason is that it is important that there is one test that is setup according to requirement specification and passing this test is the definition of completeness of the implemented design of said functionality.

Perhaps a stretch goal, but more and more becoming a reality, is that after the target hardware is released, the simulator is still used for a number of reasons that I’ll describe later, and so there will be continuous validation of the simulator against the real hardware. All test cases should have the same result on real hardware as in the simulation. But there will be situations that are beyond the simulation to do correctly, one is exact timing. The simulation we use here is loosely timed, meaning that it is acceptable with a time discrepancy from running on real hardware or on a RTL level.
simulation, which is cycle accurate, which means that all event will be correct within a clock cycle but the simulated clock will run much slower than the real clock. There will also be some tests where world time is used, meaning the input stimuli can't be synchronized with the simulated time. This is when the test equipment (or application) is not adapted for generating stimulus or responses synchronized to (much slower than real life) simulated time. Virtual test equipment that injects stimuli into the simulation has a timestamp on each event, so the simulator engine can schedule the event at the correct simulated time.

The reasons that the simulation is used even when real hardware is available are several, one is that a complete test lab with target hardware is expensive, so if a test could be done in simulator using virtual equipment it is a big saving. But more important is that some debugging can only be done in a simulator. In a simulator you can stop the time, completely, and then analyze the state of every part without worrying about losing incident information from a part of the system as you would in a real lab environment. Real hardware does not typically have access to inspection of every part, and some information retrieval is destructive because the very act of accessing the information will destroy other information. Example of something that is difficult to access in real hardware is the cache registers.
**Anatomy of the Proxy solution**

The simulator is connected to a proxy server using a SSH tunnel. The test applications and other services that should be accessible from the simulated target are connected to the proxy as well using SSH tunnels. (See Figure 13) Note that the simulator server and test server could be the same machine.

![Diagram of the Proxy solution](image-url)
Software implementation

Figure 14: The location of the first Proxy solution software components

TunnelSetup

The heart of this first proxy solution is using tunnels over an SSH session. All the necessary functions to setup the tunnels are done in the TunnelSetup utility (Runåker, 2014m). The TunnelSetup introduces the concepts of persistent sessions. From usage perspective the setup procedure is staged into first establish a secure connection to the proxy server. When the session has been setup, the tunnels can be established dynamically. It handles both tunnels that forward local clients and remote clients.
The most important contribution TunnelSetup adds to this type of tool that is missing in standard SSH (Ylonen), is the automatic finding a free local port to use in forwarding tunnels. This makes the ssh command by itself unsuitable for this proxy solution. *NOTE: Standard ssh supports dynamic socket allocation of connecting client sockets when creating forwarding tunnels, but not dynamic listening server sockets.*

One of the constraints we had to work around is that you cannot pre-allocate forwarding ports. The tunnels are created and removed dynamically as needed by the applications that are communicating with the target. The applications are not aware of the need of a tunnel to connect to the target so the tunnel creation process is totally hidden from the view of the application.

Almost all applications used for this project are written to expect there is a direct connection to the target, and have no support for any proxy connection protocol. This puts a requirement to this tool that the tunnel creation must be totally transparent and must not require a rewrite of the legacy applications that needs access to target. This means there will be numerous tunnel creations, which in turn means we need an unused socket port where the tunnel originates. But ssh command cannot find an originating free port when creating a tunnel, so this is taken care of in my TunnelSetup. The implementation of autoforward command performs a search for an unallocated socket port on the local host and then feed that information to the control port that has been setup by ssh. Ssh can by itself dynamically find a socket on remote host and report the port number back. TunnelSetup will then forward both the local and remote socket port to the next layer in the proxy solution.

Another important factor is the teardown of tunnel must be totally automatic. There are a number of situations that can cause a tunnel teardown. I implemented a remove command that did a controlled teardown, but in practical tests we found that the applications where not written to do this explicitly. The solution was to add monitoring of the application and keep track of all tunnels it uses. In that way, TunnelSetup could initiate a tunnel teardown of each tunnel when the application exits.

**Handling exceptions**

During the development of TunnelSetup several gotchas has been discovered in the underlying ssh command. First one was that multiplexing is a recent development in the ssh command so it is not available in older versions. It also has a little bit of different behavior depending on version used. I solved it by making the ssh client a local user installable command, so this was used instead when the ssh client provided by the system was too old. It was not possible to change the ssh server, but practical tests has shown so far it was enough to have a recent ssh client to get the needed feature set.

The second more sinister problem we found during testing of TunnelSetup was the stale control socket problem. Under certain conditions the secure session could go down or in other ways stop transferring data, and the control socket daemon that ssh command controls did not close the socket. I observed that killing the daemon before the session went down could create this condition. But probably there could be other situations that
made this happen. TunnelSetup did not cause this error, but it happened often enough so it had to be fixed. This situation is now detected and handled in TunnelSetup.

**TAPmanager**

TAPmanager handles the resource management of all available TAP interfaces (Runåker, 2014i). This tool manages the pool of available TAP interface and active sessions connected to the interfaces. TAPmanager also keeps track of the IP address and socket port associated with each TAP interface. The commands available are allocate, remove, port, ip and list.

The TAPmanager is implemented as a web service, so it can be accessed from remote clients easily. The results are presented as JSON (ECMA, 2013), so applications that use this tool can easily parse the result.

One important requirement is that a team can allocate a target IP address and keep it over several sessions, where the actual test target is restarted, or even replaced, so test applications where you need to manually enter target IP can be kept unchanged. The TAPmanager keeps track of who has allocated the TAP interface; so if needed, the user can be contacted if someone else need access to the TAP interface. As the resources are limited, this will be needed from time to time.

**Tapdaemon**

The tapdaemon is a simple daemon, I just made minor modification from an open source implementation called “simpletun” (Runåker, 2014g). The difference between TUN (for network TUNnel) and TAP interface is that a TUN interface only transfer the IP part of the protocol, but a TAP interface also include the Ethernet part of the protocol. If you are implementing a VPN like solution it is often enough to just forward the IP part and the Ethernet part would only be redundant with no data that contribute to the payload. But this implementation will use a number of protocols and potentially other protocols than IP. The most generic solution where the whole PDU is transferred is to use TAP interface.

The way a TAP interface works is that you get a file access to a virtual interface that connects to the TCP/IP stack on Linux (Krasnyansky, 2000). This file access interface is connected by the tapdaemon to a socket server. The information tapdaemon needs is what socket port and TAP interface it should bind to. Because failures on either side, TAP interface or socket connection, will terminate the daemon, other tools in the framework monitor the presence of this daemon to detect a network teardown.
Portsetup

Portsetup is the tool used on client side to provide and manage free local ports to be used in connecting to a specific remote port over proxy (Runåker, 2014c). It connects the TAPdaemon running on a Proxy server. The main objective was to make a scriptable tool to handle all ports used both for test application use, and for the simulation engine to connect to its TAP interface.

This means that a usage like this is possible “set IP=`portsetup -p $MANAGER_PORT -e ip`”. The output from the tool is optimized for assigning script variables. The commands handled are: allocate, remove, ip, port, list. It also decodes the JSON response from TAPdaemon so it extracts only the information that is requested.

Extee

Extee is an extended tee tool (Runåker, 2014a). The need of this tool grew from a need to totally automate the testing of the target. I added this tool to the framework because there are a specific use case that need this tool. It is when one application needs to connect to the telnet port of the target’s console during boot-up. The simulator allocates the telnet port and it is not known which port it will use. On this server several instances of the simulator is started simultaneous and each one will get a unique port. The information about which port the simulator chose to use is printed as terminal output. What was needed was to find a specific string that was written to the simulator log file and then extract the port number, and from that information start the application with the right parameters so it connected to the console. The application was part of a Continuous Integration framework that used Jenkins to do regression tests on each software package delivery. This application was used on real hardware and it should not be changed when used on the simulated target. The port they needed is outside test network that the proxy solution handled, it is instead provided by the simulation engine directly and could be seen as a serial interface not a network adapter.

After reviewing the available common tools, I decided it was better to write it for a number of reasons. It should behave as the tee command, detect a pattern without unnecessary buffer delay, use regular expression both for pattern matching and extraction, build a command line with variable instantiating, start background process using this command line and be able to pipe together several extee.

The reason we did not modify the simulator engine to provide this information directly was that the part of the simulator that allocated this port resides in ARM’s FastModel, a loosely timed simulation of ARM processor (ARM, 2015), and I did not have access to the source code.
setIPfromMac

The setIPfromMac sets the IP address according to MAC address (Runåker, 2014e). This is a script that is needed on the target. Each target needs to have a specific IP address when it starts. The dhcp client typically handles this, but the test net we have currently available did not have access to a dhcp server. The simulator configures the MAC address from a configuration file. A script modifies this file before the simulator is started so it gets a unique MAC address and also information that the setIPfromMac can use to set the correct IP address.

Tsocks

This is a clone of the official tsocks library, with specific build instructions so it works with TunnelSetup (Runåker, 2014k). I have been working with moving all dynamic tunnel setup to use SOCKS server instead (Leech, 1996). More specifically the SOCKS server provided by ssh (Lee).

There are multiple benefits by using SOCKS when connecting to the target. First there is no need to setup tunnels before the application is started. Even more important is that the applications can use the original IP address and port number. And perhaps most important is that applications that create connections that are not known beforehand cannot be handled with TunnelSetup in manual mode.

If you use the –s parameter to TunnelSetup, you activate the SOCKS server. By switching to SOCK, the application can connect to the target as if it had direct connection to it. But most application does not have built-in support for connecting using SOCKS protocol. This was my stretch goal to get even these legacy applications to use SOCKS.

The solution I choose was to use tsocks. This application uses a technique called library injection. To get this to work for applications without SOCKS protocol support, it means that the application needs to be “socksified”. This means it needs a LD_PRELOAD on the command line before the command that starts the application. Tsocks replace or injects a special version of the standard library functions for networking. What is happening is that, when the application connects to a socket using GNU C Library connect function, it actually use tsocks version instead. Tsocks will execute the necessary SOCKS protocol exchange to the SOCKS server and open a tunnel to the target, and then let the original connect command go through that tunnel.
TunnelingRecursiveRouter

The current solution works very well and has good performance. But there is one component that limits the scalability of the solution, and that is the reliance on the ssh command. The culprit is that every user that need access to their simulator instance, need a login on the proxy server so ssh can be used to create the tunnels. The IT department made it a requirement that users of the proxy solution should not need account on the Proxy server, or they will limit the number of users that get access to the proxy solution.

This is a good challenge, because this is an opportunity to make a number of improvements. First I wanted something that was a bit more lightweight than a complete SSH session setup (Stallings, 2009). This would reduce the latency when starting the initial session, and also for each tunnel created. The other performance problem is when the number of SSH sessions becomes huge; it means that the solution becomes less optimized. One reason is the memory requirement for each session, and the other is the processes that are created for each session just to wait for data to transfer require too much processor power.

We need to protect the tunnels cryptographically, so it is secure from intentional and unintentional data leakage. It means that it is protected from packet sniffing and man-in-the-middle attacks.

The next version is built on a shadowsocks using libev. The official version is not useable for this, so I have modified it so it works with the Proxy solution (Runåker, 2015a). Shadowsocks use libSSL instead of SSH and totally configurable of which crypto to use (even null crypto). I choose the libev version because every session it sets up uses very little resources. It is optimized for virtual servers where memory footprint and processor utilization are very important factors.

It is expected that many times more sessions can be created with the same resource utilization as if using SSH. The next version is also much more ambitious than the current version so it handles more complex topologies where an arbitrary number of Proxy servers can be between simulation target and test applications. It also implements support for UDP and tunnels that originates from target back to any service or application on the test net.

We call this implementation TunnelingRecursiveRouter (Runåker, 2015c), or TRR. TRR handles situations where extreme complex network solutions are required. It builds an infrastructure of multilevel encapsulated networks that gives a transparent bidirectional VPN-like connectivity where different nodes are located in tightly locked down secure environments. It supports TCP and UDP over IPv4 and IPv6, and optionally all other protocols by using raw packets. By using a static reverse tunnel for each listening socket, the target on the other side can do an ordinary connect to this service without changing the client code, Figure 15.
The proxy solution version 2 (TRR) diagram

Figure 15: Simple TRR setup
Connecting two different virtual nets using TRR

As seen in Figure 16, by using two TRRs we can connect two virtual nets and make it routable to each other even if they reside on different servers.

Figure 16: Two TRRs connects two virtual net
Connecting a service from one net using a tunnel

This is a more generic case and scales much better because each virtual net can be run on different hosts. Because we are running without any privileges on the server we got a problem with routing between the virtual nets when we can’t do the necessary changes to the routing table. This can be solved in a number of ways. The simplest solution is to make a socket tunnel on each virtual net, so there is no need for routing, because the service is available on the same net as the client. Note that the IP address for each service accessed will be on the same net as the client is residing on. You can see this in Figure 17, where the IP address of the service will be in Net 1, and TRR creates a tunnel to its own client on the other net where it can connect to the service.

![Diagram of client and service on separate nets connected by a tunnel](image)

Figure 17: Accessing a service on a different net

This is a good solution when the reason for using several virtual nets is to increase performance by scaling to more hosts. But if you really want to place the services on a different net to make it more similar to real network situation, this is not an acceptable solution.
TRR based routing between different virtual nets

![Diagram of TRR routing](image)

Figure 18: TRR handles routing based on location of service

Figure 18. Here the service opens a reverse tunnel to the net it should reside on, in this case Net 2 via SOCKS server 4. The client connects using the port and net number where the service is located. Because there are two TRR proxy handlers running it will dynamically choose the right one based on the network number, which is Net 2. From the client point of view, it will use the correct network address and TRR will take care of the actual routing without using the routing table of the host system. If the client do not handle SOCKS protocol, it will be necessary to use a library injection of, for example, tsocks, which is already used in proxy solution version 1 (Runáker, 2014k). The configuration file for each local TRR instance contains the IP address range it can connect to, so it will only setup a tunnel if that address is on the other side. NOTE: It is important to make sure that there is no overlap of IP addresses on the different nets that TRR connects to, so at most one connection is made for each connection request over TRR.
No target impact, all TRR handling outside of target

The stretch goal is to remove the need of placing the TRR proxy handler and tsocks injection on the target, and place it on the proxy server where the target is connected. The aim is that no extra code should be added to the target because of the simulation. This is doable, because the simulated target is connected via a socket and then to a TAP device. As you can see in Figure 19, the raw packets can be inspected outside the simulation and routed based on destination address before sent to the TAP device.

![Figure 19: Routing outgoing traffic from target without adding SOCKS client inside simulation](image_url)

The solution is to make the simple socket server for the TAP interface more intelligent. This involves packet inspection so it extracts the IP destination address, and route the packets that are going to a different network to the TRR handler. Each new IP+Socket destination will create a new tunnel to the other Proxy server that handles the destination network and from then on automatically be forwarded into this created tunnel.
Now we don’t need to create static tunnels before the simulation, and the TRR solution will be totally transparent from target point of view, because TRR will be outside the target.

The tunnel will be available as long as the TCP connection is valid. If it is an UDP connection, the tunnel will be removed when no traffic has been transferred for a while. If there is more UDP traffic, the tunnel can be setup without interfering with the transfer because there is no handshake sequence, which is needed in TCP protocol. *NOTE: when a network condition creates a TCP teardown, the TCP client must be notified so it can set it up again.*
Increase performance by moving execution out of target

If it were possible to execute part of the application outside the simulated target it would be much simpler to increase performance. We have a perfect opportunity in this project to do just that! A major part of the execution time is used by something called middleware. This heavy weight of software is written in Erlang (Ericsson, 2015c), a language not only developed at Ericsson but also the lead developers of the language were located in the same corridor where I was working on the Proxy solution. During one lunch with the members of the Erlang OTP team (Ericsson, 2015e) I was discussing the problem with the Middleware and the Erlang interpreter. The problem is that one interpreter is spawned on each core in the system. This would be efficient if the core where physical but here they were simulated and no parallelism was achieved by spawning on several cores. The FastModel ARM simulator has only one thread that runs each ARM core in turn, so applications that utilized parallel execution on several cores in the simulated ARM, ran slower than a single threaded application that utilized just one core.

The result was even worse when considering that the Erlang application, or rather a byte-code representation of it, was running in an interpreter (a process called beam) that in turn was running in a simulation of an ARM cluster and to make the matter worse, everything was typically running inside a XEN virtualization on a Linux server. This means there are several layers that reduced the performance of the application. I learnt that Erlang have several ways to scale and use available computational resources.

When Erlang OTP team learnt about my Proxy solution and the possibility to connect the simulated target to any real service on the simulated network, they described how to configure the Erlang environment to utilize external CPUs without changing the application. One of the important constraints of the simulator is that exactly the same binary will run on the simulator as on the real hardware. But we were allowed to use different DTB. The Device Tree Blob is like a database that describes the hardware platform. It contains the devices and their hardware configurations (e.g. memory addresses, port number, interrupt number, etc) that are read by U-Boot and Linux during early initialization phase, but it can also contain custom information that can be parsed by the applications running on target. This is the place where we were allowed to store the special configuration for the middleware application. This makes it possible, when running in a simulator, to configure the runtime environment for the Erlang application so it will setup a network connection to another instance of the middleware that is executed on an x86 server. On the target all requests to the middleware application will be forwarded over the virtual network to the instance running on the x86 server. The instance running on x86 server can use all available cores and, if running on a high-end server, actually execute the middleware application much faster than on target.

The Erlang applications will not execute slower than on target when running in the simulation anymore, it will actually run faster! Because this Erlang application represents the majority of the execution time of the simulated system, the overall system performance will therefore improve. The application developers that write code in Erlang will not be aware of any differences when running on hardware or on simulator. Just to restate what is now obvious; the differences are abstracted away into
the DTB, which is unique for each target platform, and the simulator is just one of the target platforms.

This is on the simulated target, but the native host that is actually running the Erlang application will require some additional framework. This code is outside the application created in this project so we still fulfill the requirement of same binary on simulator and real hardware when not considering a platform specific version of external libraries that resides on hosts outside the simulation.

One of the changes we have identified is the file access done on the native server needs to be changed so it tunnels back to the file system on the simulated target. All the changes necessary to be done are located in the Erlang OTP layer. Although the implementation of this is still in progress, it is considered to be straightforward.
Conclusions

In this paper I have presented how to find a solution to increase performance of the simulation by offloading some of the execution from the simulated target. There was also a short overview of the TLM based simulation that was used at Ericsson, and the performance problem around simulation in general. It was also shown how certain performance issues were solved.

The main topic for this paper is the proxy solution, because it was necessary to enable networking in the simulated system. The first implementation of proxy solution has been running for more than a year. It has been used by hundreds of people in different teams at Ericsson, and also as a part of the test suite running on continuous integration servers (Jenkins) that runs around the clock testing every software change before merging it into next release. The solution is proven to be robust, and no issues are known with the current implementation. There are several works in progress, for example the roll out, and implement some more special use cases, of version 2 of proxy solution (called Tunneling Recursive Router) and the implementation of performance improvement of the Erlang applications.

Future

Next generation of the simulation platform has very ambitious goals, and will simulate even larger part of the base station and also include, for example, the radio unit. It connects to a larger number of parallel simulation engines than before, and the CoSim/EMA connection between the different simulation engines will be extended to handle even more simulation events (probably using features from the upcoming CoMix adapter that Ericsson already use to communicate with the radio unit simulation). Ericsson has communicated that the importance of simulation of the target for use by software developers is very high, and will be used in all future systems of this type. Further more the test labs are starting to move out into the cloud so will the need of complex topologies of virtual network, which will require solutions like TRR. The idea is to create a simulated test node on demand from anywhere, complete with specific version of a base station and complete with test equipment, all virtual and instantaneous. This will lessen the need for physical test labs for software testing. The development process is about to change a lot just within a year!
References


Ylonen, T. ssh — OpenSSH SSH client (remote login program). from http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man1/slogin.1?query=ssh&sec=1
