Design of an Event-based Server

Bachelor Thesis by Peter Schuster
January 2013

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Electrical Engineering and Information Technology (ETiT)
Institute of Computer Engineering
Integrated Electronic Systems Lab
Prof. Dr.-Ing. Klaus Hofmann

Herewith I declare, that I have made the presented paper myself and solely with the aid of the means permitted by the examination regulations of the Darmstadt University of Technology. The literature used is indicated in the bibliography. I have indicated literally or correspondingly assumed contents as such.

Darmstadt, January 2013

Peter Schuster

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project Context and Objectives

Context of this project is ongoing research for high performance network interface cards at the Integrated Electronic Systems Labs by Boris Traskov, also supervisor of this project.

In an environment with an ever increasing amount of created data by all kinds of devices, the amount of transferred data over networks (i.e. the internet) increases dramatically. This generates a demand for high speed network interfaces, on the one hand and high performance, low cost full stack network implementations on the other hand.

Service providers and back-end nodes need to be able to transfer large amounts of data to multiple receivers concurrently, requiring high speed network interfaces. To circumvent – potentially unnecessary – Central Processing Unit (CPU) utilization, much of the work could be "offloaded" to a network interface, providing a higher level of abstraction to the transfer of data, than current state of the art systems provide.

In a future scenario with all devices being connected to each other, often described as the "internet of things", an increasing demand for simple to implement network interfaces, not requiring the presence of an operating system or high performance processors, will be created.

Objectives for the whole project entitled "Design of an Accelerated Event-based Server" is to setup a hardware system on a *Xilinx* Field Programmable Gate Array (FPGA) utilizing Intelectual Property (IP) cores, running *nginx* (an event-based web server) on top. Furthermore benchmarks have to be conducted, to prove the strength of such a system. The ideal outcome of this project would be to prove that the system is capable of utilizing the full bandwidth of a Gigabit Ethernet interface. Therefore break downs of the server system on attempted Denial of Service (DoS) attacks would belong to the past.

The first part of this project included the design of a System on Chip, as well as getting an operating system up and running on top of it. This was implemented in a preceding project seminar, May to September 2012. Reached accomplishments of this project seminar are documented in the corresponding report [18]. All results are also briefly summarized in chapter 3 of this thesis.

1.2 Hardware Platform and Tool Set

All development, measurements and tests were done on the *Xilinx XUPV5-LX110T* evaluation board.

The *Xilinx XUPV5-LX110T* evaluation board is a modified version of the *ML505* board for universities [21]. The difference between the two boards is a larger FPGA chip on the *XUPV5-LX110T*, containing the *Virtex-5 XC5VLX110T* with 17,280 slices[1] and four in hardware implemented Ethernet Medium Access Control (MAC) cores, whereas the *ML505* board contains the *XC5VLX50* with only 7,200 slices and no Ethernet MAC cores implemented in hardware.

Building the hardware design for the FPGA was started using the *Xilinx Platform Studio (XPS)* and continued with the *Xilinx ISE Project Navigator*, for a finer control over the parameters of used tools and better logging and reporting capabilities. All used *Xilinx* tools are part of the *Xilinx ISE Design Suite* in version 14.1. The synthesis tool (xst) of this version of the tool suite features parallel synthesis speeding up the development process heavily.

Initial software development for testing purpose was done using the *Xilinx Software Development Kit (SDK)*. The *Linux kernel* itself and all software build on top was configured and compiled on a virtual machine running *Ubuntu Linux* in version 12.04.

[1] Basic logic unit of an FPGA [13]

2 System Requirements

Requirements for the designed system must be collected with two key points in mind: the desired outcome of the project and the chosen/available hardware system and tool set.

Objective of the project is to run *nginx*, an event-based web server. *nginx* can not be executed directly on the processor, because it requires the presence of an an Operating System (OS) providing a file system and taking responsibility of process and thread management.

nginx supports a number of free (FreeBSD, Solaris, Linux) and proprietary (AIX, HP-UX) unix-based operating systems, as well as *Microsoft Windows*.[1]

Microsoft Windows is currently only available for Intel's *x86* and AMD's *AMD64* (also known as *x86-64*) architectures. The constraint on the available *Xilinx XC5VLX110T* FPGA demands an easy-to-implement, low cost microprocessor system. But the mentioned processor architectures do not count towards these categories. Therefore the choices are limited to free unix-based operating systems.

Due to the by far largest number of supported processor architectures and wide distribution, the choice was made to go with Linux as operating system for the project.

Linux demands a Memory Management Unit (MMU) in virtual mode and two memory protection zones.[2] MMUs enable an OS "to exercise a high degree of management and control over its address space and the address space it allocates to processes" [10][sec. 2.3.5]. Linux kernel furthermore requires the presence of two timers and an interrupt controller.[3]

The used web server software (*nginx*) requires the availability of an event module in the OS. Linux kernel contains a number of event modules, all being supported by *nginx*. The preferred and most effective one is *epoll*.[4]

[1] see `http://nginx.org/en/#tested_os_and_platforms`

[2] Linux kernel can be configured for processors without MMU, but this is not recommended.

[3] see `http://wiki.xilinx.com/microblaze-linux#toc4`, as of 09/07/2012

[4] `http://wiki.nginx.org/NginxOptimizations` (as of 12/2012)

3 Preceding Work

Preceding to this bachelor thesis, a project seminar with the same title was taken. During this project seminar the first part of the project was implemented.

The outcome of this project seminar is described in the following chapter. A detailed description on how this was accomplished can be found in the respecting project report [18].

3.1 Hardware System

3.1.1 Architecture

In this project a System on chip (SoC) was build on top of an FPGA using predefined IP cores for the processor and additional system components.

An overview of the complete hardware architecture of the system described in the next sections is included in the appendix of the project seminar report (see [18], sec. A.1).

Processor

The used *XC5VLX110T* FPGA does not have a build-in hard core processor, therefore a single core *MicroBlaze* soft core processor was chosen. The *MicroBlaze* processor is a proprietary processor, developed by *Xilinx* for their FPGA families and supported by the Xilinx hard- and software development kits. Its design follows the Harvard architecture with separate data and instruction memory.

Running Linux kernel requires the presence of an MMU. To improve the system performance instruction and data caches (16 KB), barrel shifter, multiplier (64 bit) and the hardware division modules were enabled.

Bus System

For connecting the *MicroBlaze* processor to other peripherals on the chip the Processor Local Bus (PLB), invented by *IBM* as part of the *CoreConnect* bus system, was selected.

Prior to the *Virtex-6* FPGA family, only this bus system was available. *Virtex-6* FPGAs support also the Advanced eXtensible Interface (AXI) system, which is part of the Advanced Microcontroller Bus Architecture (AMBA), designed by *ARM*. The used *Xilinx XC5VLX110T* FPGA is part of the *Virtex-5* family, therefore PLB needed to be selected as interconnect type. [25][p. 1, facts table]

Memory

The used evaluation board of type "*XUPV5-LX110T*" contains a single-rank unregistered 256 MB DDR2 SODIMM, which is connected to the processor via a memory controller. This memory controller is implemented in the *Multi-Port Memory Controller* (MPMC) IP core. The memory base address was set to `0x50000000`.

Network Interface

The *Xilinx XC5VLX110T* FPGA has four *Tri-Mode Ethernet Media Access Controllers*, designed to the IEEE 802.3-2002 specification, operating at 10, 100, and 1,000 Mb/s. [22][p. 4, table 1] To use these hard core controllers an `xps_ll_temac` soft IP core was added to the SoC, acting as a wrapper for the hard core to integrate it into the system.

Gigabit Media Independent Interface (GMII) is a backwards compatible extension to Media Independent Interface (MII) supporting data rates of up to 1,000 Mb/s. Therefore it was selected as physical interface type, because support for Gigabit Ethernet was desired, but there was no need for a reduced data path width. To enable correct operation mode using GMII as physical interface type on the *Xilinx XUPV5* board, the jumpers `J22` and `J23` need to be set to positions `1-2`.

Usage of an integrated checksum calculation circuit is enabled using the parameters `C_TEMAC0_TXCSUM` and `C_TEMAC0_RXCSUM`.

3.1.2 Clocks

Clocks for the system are generated using a *clock_generator* IP core with an external oscillator providing a 100 MHz clock. [24][p. 20]

Due to high delays on data paths in the decode pipeline stage, a clock period of at least *9.12 ns* is required, resulting in a system clock frequency for the processor and local bus of 100 MHz.

The memory controller (MPMC) is driven by a base clock of 200 MHz, a clock with half the frequency of the base clock (100 MHz) and a 200 MHz clock signal, shifted by 90°. All these clock signals are controlled by the same phase-locked loop (PLL) used by the system clock signal.

The `GTX_CLK` port of the Ethernet MAC IP core is driven by a clock signal with exactly 125 MHz for operating *GMII* (defined by the specifications of GMII [1][sec. 35.2.2.1]). For Direct Memory Access (DMA) and control logic, a clock signal with a frequency identical to the local bus clock is required. The `REFCLK` was connected to a 200 MHz clock, according to the respective manual of the IP core [23][p. 11, table 3].

3.1.3 Endianness

> "*Endianness describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system.*" *[5][p. 5]*

Architectures utilizing the little endian concept store the least significant byte (LSB) at the lowest address, in big endian architectures the most significant byte (MSB) is stored at the lowest address. [5][p. 6]

Linux can be build for little, as well as for big endian systems. Only confinement is that the used toolchain (compiler, etc. – see [18][sec. 3.2.2]) needs to support the endianness of the architecture.

The *MicroBlaze* processor has the parameter `C_ENDIANNESS` to specify the endianness of the processor. But although the *MicroBlaze Processor Reference Guide* states that "the `C_ENDIANNESS` parameter is automatically set to little endian when using AXI4, and to big endian when using PLB, but can be overridden by the user" [26][p. 52], this parameter must not be changed for *Virtex-5* FPGAs. This is reasoned in the disability of the peripheral cores connected via PLB, to handle data other than in big endian byte order. The AXI bus circumvents this problem by swapping bytes.[1]

Therefore big endian was selected for the system architecture of this project.

[1] `http://forums.xilinx.com/t5/EDK-and-Platform-Studio/Memory-Test-fails-for-8-and-16-bit/m-p/253922/highlight/true#M23973` (in-official statement by a Xilinx employee)

3.2 Software

3.2.1 Linux Kernel

Linux kernel was chosen as the OS for the system. This is the pure core Linux OS, in comparison to enriched Linux distributions (like *Ubuntu*, *Debian*, *openSUSE*, *Fedora* and many more), which contain additional libraries, applications and configuration. The Linux kernel version used in this project contains further additions and bug fixes specific to the *MicroBlaze* architecture by *Xilinx*.

To enable correct recognition of the latest *MicroBlaze* processor versions with enabled Processor Version Register (PVR), it might be necessary to apply a patch included in the appendix of this report (B.1.2) to the Linux kernel sources. The patch was extracted from the *PetaLogix* Linux kernel fork.

Building Linux kernel sources to an executable image file requires a set of tools (called toolchain) for compiling source code files and linking binary output to one single image. *Xilinx* provides toolchains, based on the widely used GNU Compiler Collection (GCC) and *binutils*, for cross compiling from Linux x86 and x86-64 architectures to *MicrobBlaze* systems as target architecture. Depending on the used development system, the corresponding toolchain has to be selected.

Configuration

Linux kernel consists of many optional sub-parts for target architectures, device drivers, special features, etc. Which of these parts are compiled and linked into the Linux kernel binary image needs to be configured in the `.config` file in the Linux kernel root directory. This file is "the configuration blueprint for building a Linux kernel image" [10][sec. 4.3.1] and contains all (required) settings.

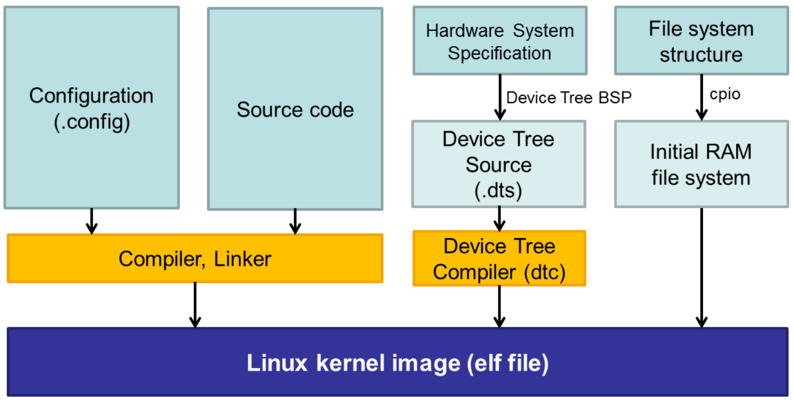

Figure 3.1.: Components of a Linux kernel build.

The *MicroBlaze* processor can be configured with different feature sets (multiplier, barrel shifter, etc.). Therefore the GCC compiler needs to be parameterized for matching the provided features of the target system [27][sec. "Kernel Configuration Details"]. These need to be specified in the `XILINX_MICROBLAZE0_*` settings inside of the `.config` file. The `KERNEL_BASE_ADDR` is also an imported setting which must match the base address of the system's main memory.

The `XILINX_LL_TEMAC` driver is used for the Ethernet interface.

Another part of system configuration is the *Device Tree*. It is an abstraction layer for accessing hardware information, to avoid compiling everything into architecture specific assembler code [8]. It is accessed by the Linux kernel during the boot process for configuring itself and on lookup of hardware information. Therefore the *Device Tree Source (dts)* file is compiled by the *Device Tree Compiler (dtc)* during the Linux kernel build process to an *Device Tree Blob (dtb)* and linked into the final Linux kernel image. It can be generated using the *Device Tree Generator*, a Tool command language (TCL) script reading a system specification file generated by XPS.

3.2.2 The File System

It is possible to use the Linux kernel without a file system, but the makes little sense in practical use. Therefore an *Initial RAM Disk (initrd)* image is mounted as *root file system (rootfs)* [10][sec. 6.1]. This is a file system packed into a *cpio* archive and linked into the Linux kernel image. It is unpacked completely into the main memory during kernel boot process.

3.3 Deployment

The generated *bitstream* of the hardware system representing the SoC, needs to be programmed into the FPGA on every power-on. This is required, because the configuration of the FPGA being set by the *bitstream* is volatile. Programming the *bitstream* is straight forward and can be accomplished using the *Xilinx iMPACT* tool, which is part of the *Xilinx ISE Design Suite*.

When the Linux kernel build succeeded, the resulting *Executable and Linkable Format (ELF)* file can be found at `arch/microblaze/boot/simpleImage.<dts-name>`. This file needs to be loaded into the FPGA using *Xilinx Microprocessor Debug Engine (XMD)*, which is also part of the *Xilinx ISE Design Suite*.

4 Modifications to the System

The preceding sections describe the state of the hardware and software system, at the end of the project seminar. Although the part of the project implemented during this bachelor thesis builds on top of the project seminar's outcome, some modifications needed to be made to these parts of the system. These modifications are described in the following sections.

4.1 Hardware

Data and instruction caches were extended to 64 kilobytes. This gives a slight performance gain for the system, while the used FPGA had sufficient resources left.

4.2 Software

The toolchain used during the preceding project was updated to the latest version provided by *Xilinx*. This version has the identifier "`microblaze-unkwnown-linux-gnu-....  (crosstool-NG 1.14.1) 4.6.2 20111018 (prerelease)`". The GCC version included in this version is `4.6.2`. This GCC version is not compatible with the Linux kernel version used previously. Therefore the Linux kernel was updated to version 3.5.0 (commit `45b74487f57324fa66da40cd4d52be6f07e2aefd`[1]). The toolchain upgrade itself was required for the work on user space applications described in the following chapters.

The used web server software (nginx) requires the availability of an event module in the OS. Therefore `epoll` was activated in the Linux kernel configuration (`.config` file). This is done by setting the switch `CONFIG_EPOLL=y`. Further explanations on this change can be found in the related chapter about the nginx architecture (see 5.2).

Another event, affecting the software part of this project was the corporate takeover of *PetaLogix* by *Xilinx* in late August 2012 [7]. In consequence the efforts of both companies on Linux kernel development for the *MicroBlaze* architecture were bundled, leading – amongst others – to a common source code repository for related developments. This alleviated working on the software part of the project.

4.3 The File System

For deployment of user space applications to the test system a number of *C standard library (libc)* components need to be copied to the system. This results in a combined Linux image size of about 25 MB. Loading this image into the *MicroBlaze* memory using XMD takes about 9 minutes. For frequent develop-deploy-test cycles this adds unnecessary burden to the overall development process.

To circumvent this problem during development on user space applications, the file system was embedded using Network File System (NFS). When using NFS, a directory on another system is mounted as root file system on the *MicroBlaze* system over the network.

To enable this change for the *MicroBlaze* system, the Linux kernel configuration option `CONFIG_CMDLINE` needs to be changed to the following value:

```
CONFIG_CMDLINE="console=ttyUL0 ip=192.168.2.125 rootfstype=nfs root=/dev/nfs rw ↩
nfsroot=192.168.2.112:/home/peschuster/customfs/complete,tcp"
```

[1] `http://git.xilinx.com/?p=linux-xlnx.git;a=commit;h=45b74487f57324fa66da40cd4d52be6f07e2aefd`

This sets `192.168.2.125` to be the static IP address, changes the root file system type to NFS, connects to a NFS server at another system with IP address `192.168.2.112` and mounts the directory `/home/peschuster/customfs/complete` as root file system with read and write permissions over TCP. It might be required to adjust the exact values for theses settings, depending on the current environment.

On the system hosting the mounted directory, NFS server needs to be configured for allowing the respective directory to be mounted by the other system. On *Debian* and *Ubuntu* Linux distributions this is done by adding a line to "`/etc/exports`", which should look like the following:

```
/home/peschuster/project/customfs/complete 192.168.2.125(rw,sync,no_root_squash)
```

This allows access to `/home/peschuster/project/customfs/complete` for mounting by a system with IP address `192.168.2.125` with read and write access and synchronous data transfer. The option `no_root_squash` allows access and modifications to files and directories belonging to the `root` user. It is required, because there are no users and user groups defined on the *MicroBlaze* system.

For conducted performance tests in a later stage of this project, these temporary changes to file system mounting were reverted and the file system was embedded into the Linux kernel image as a packed *cpio* archive.

5 nginx

5.1 Introduction

The context of this project is to implement a testbed for a *TCP Offload Engine (TOE)* being developed in another research project. This testbed should enable testing the TOE in an environment and with workloads close to usage in real world. One usage of the Transmission Control Protocol (TCP) is as transport layer for Hypertext Transfer Protocol (HTTP) requests. HTTP is the second most used protocol among all internet traffic, with a share of about 20 to 30 % (in 2008/09) [11] and therefore very relevant outside of lab environments.

The server-side endpoint for HTTP traffic is a web server software. For this project *nginx* (pronounced "engine-x") was chosen, which follows an asynchronous architecture. It was released to the public in 2004 and focuses on "high performance, high concurrency and high memory usages" [2]. Creator and main developer of nginx is Igor Sysoev[1]. During the course of this project nginx versions 1.3.4 to 1.3.10 were used.

A server system, like a web server, needs to be non-blocking to handle multiple requests at once. That is it must be able to accept and process requests while it is still busy with other, previously received requests. Systems with a "traditional" (i.e. thread-based) architecture fulfill this requirement by spawning separate processes or threads for incoming requests. An application designed by this model does not favor performance, because spawning new processes or threads "requires preparation of a new runtime environment, including allocation of heap and stack memory, and the creation of a new execution context" [2].

In contrast to this, *nginx* was designed following an asynchronous architecture, which is accomplished by using a so called event-pattern. This means – very much simplified – that *nginx* never "waits" during processing a request for any external operation to complete, but pushes it to an event system, being part of the OS, does something other useful like processing new requests and picks up the event, when it is finished for further processing steps.

5.2 Architecture

The following sections provide a brief overview how *nginx* and especially the event-driven architecture work.

5.2.1 Overview

A running *nginx* instance always consists of atleast two processes: a **master** and a **worker** process. The master process spins-up, monitors and controls the worker processes. The worker processes handle the actual (HTTP) requests on a single thread.

After initialization both processes run in a so called **process cycle**. That is an infinite loop with the process waiting to be activated from suspension by external events. For the master process these are signals send to the process which might indicate requests for shutdown, configuration reload or previously initiated timed events.

Design of the worker process had as key principle "to be as non-blocking as possible" [2]. This is done by relying heavily on asynchronous operations being implemented through "modularity, event

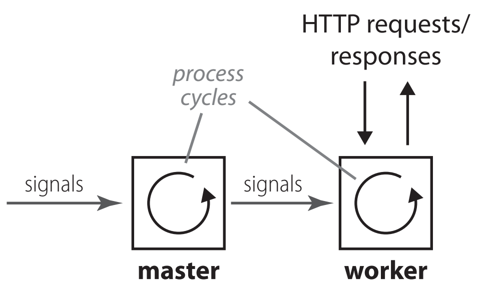

Figure 5.1.: Overview of nginx process model.

[1] http://sysoev.ru/en/

notifications, extensive use of callback functions and fine-tuned timers" [2]. It also makes the process cycle of the worker process the "most complicated part of *nginx*" [2].

Processing of requests is outsourced from the *nginx* core routines to a number of dedicated modules. These modules hook into a processing pipeline forming a **chain of modules**. When a request receives, it is passed through the pipeline with every module doing the relevant work [2]. Functionality of modules includes handling a specific protocol (e.g. HTTP), modifying content, filtering, handling special variables or load balancing. It is also possible to develop and integrate custom modules.

An HTTP request runs through a number of processing phases with dedicated **phase handlers**. These process a request and generate an appropriate response, send the header and send the body. Generation of content is done by **content handlers**. Out-of-the-box there exist several default handlers for index views or simple static files. The result of this handlers is passed to **filters** performing outbound content modifications. Filters form a separate processing pipeline, passing results among themselves until the final filter is called. Functionality of filters includes amongst others generation of header data, charset modification and gzip compression. [2]

5.2.2 Event-driven Architecture

Another task of modules in the *nginx* architecture is to implement functionality specific to a certain OS. One of this specifically designed tasks is the integration of an event module. Consistent and strict usage of event-driven architecture is the fundamental point of *nginx*.

This is accomplished by using asynchronous, i.e. non-blocking, i/o-functions of the operating system. Handling of returning events is done through an event module provided by the OS. For this project Linux's *epoll*[2] module is used. *epoll* handles events on file descriptors. Therefore also network related and custom events can be handled, because in Linux basically everything is a file descriptor[3] (or process).

Events are queued and dequeued in the worker process cycle. The total workload for processing a request is split into multiple chunks, each handled after preceding, necessary operations of the OS returned.

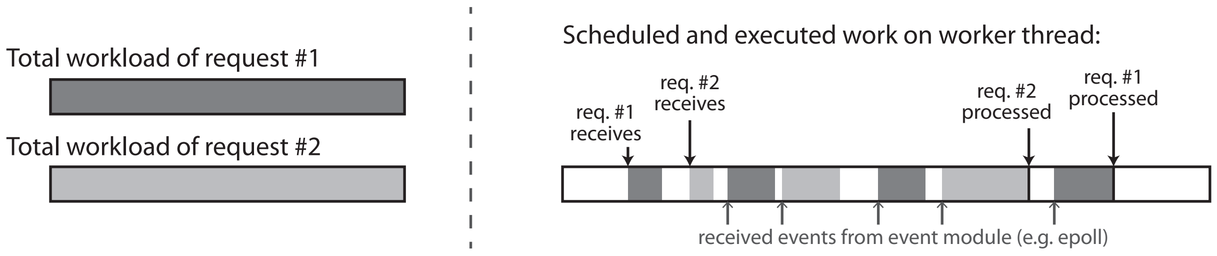
Total workload of request #1
Total workload of request #2
Scheduled and executed work on worker thread:
req. #1 receives
req. #2 receives
req. #2 processed
req. #1 processed
received events from event module (e.g. epoll)

Figure 5.2.: Processing model of worker threads.

This leads to a responsive nginx worker thread which "can handle many thousands of concurrent connections and requests per second" [2] (on a decent server system).

2 `http://linux.die.net/man/4/epoll`

3 Linus Torvalds, "The everything-is-a-file principle": `http://yarchive.net/comp/linux/everything_is_file.html` (as of 12/2012)

5.3 Configuration and Building

After this general overview of *nginx*'s architecture and underlying concepts, the next section focuses on practical implications of bringing *nginx* to a *MicroBlaze* system.

5.3.1 Extending the Configuration System

The build process of *nginx* can be configured with a number of parameters and constants to add or remove optional modules. The inclusion of modules can be configured with command line arguments to the configuration tool, but many other parameters can not be set externally.

The values of these parameters are determined by a custom "auto configuration" tool. This tool writes small C programs to a temporary file, compiles them using the configured compiler and reads back the results. By this process *nginx* adjusts its own build process to the features and properties of the current system. This configuration process is not working for cross-compiling *nginx* for another target system. Therefore the configuration process needed to be extended to allow input of required configuration parameters into the build process from an external source.

The implemented solution is based on a suggested patch by Daniele Salvatore Albano[4], but incorporates more of the existing parameters and is streamlined to the process of cross-compiling the Linux kernel. Cross-compiling can be enabled by setting the environment variable "`CROSS_COMPILE`" to the suitable compiler tool chain prefix. For the *MicroBlaze* tool chain this is "`microblaze-unknown-linux-gnu-`".

Parameters that are covered by this modification to the configuration system include endianness (`with-endian`), the size of primitive data types (`with-int`, `with-long`, etc.) and the maximum error number used by the OS and therefore not to be used by custom error types (`with-sys-nerr`).

To determine correct values for these introduced parameters, a small test program was written which acts in a similar way like the original nginx-auto-configuration tool (see appendix B.2.1).

Executed on the designed SoC, the program output leads to the following values:

```
--with-endian=big \
--with-sys-nerr=132 \
--with-int=4 \
--with-long=4 \
--with-long-long=8 \
--with-ptr-size=4 \
--with-sig-atomic-t=4 \
--with-size-t=4 \
--with-off-t=4 \
--with-time-t=4
```

5.3.2 Debug Mode

By specifying `--with-debug` for configuration of *nginx*. Logging and calculation of additional debug output is compiled into the resulting binary image. This is good for an initial setup and test phase, but was disabled for performance benchmarks, conducted at a later stage of the project.

[4] *Cross compilation support for nginx*, Daniele Salvatore Albano, 01/03/2011 `http://web.archiveorange.com/archive/v/Tuw7Ryz8rztiNaIFfqCg`

5.3.3 Modules

nginx consists of a number of (optional) modules. These modules can be in- or excluded from the build using parameters to the configuration tool. Objective of the *nginx* build for the MicroBlaze system was a small binary with just the necessary parts included. Therefore the following modules were excluded:

```
    --without-http_rewrite_module \
    --without-http_gzip_module \
    --without-http_charset_module \
    --without-http_ssi_module \
    --without-http_userid_module \
    --without-http_access_module \
    --without-http_auth_basic_module \
    --without-http_autoindex_module \
    --without-http_status_module \
    --without-http_geo_module \
    --without-http_map_module \
    --without-http_split_clients_module \
    --without-http_referer_module \
    --without-http_proxy_module \
    --without-http_fastcgi_module \
    --without-http_uwsgi_module \
    --without-http_scgi_module \
    --without-http_memcached_module \
    --without-http_limit_conn_module \
    --without-http_limit_req_module \
    --without-http_empty_gif_module \
    --without-http_browser_module \
    --without-http_upstream_ip_hash_module \
    --without-http_upstream_least_conn_module \
    --without-http_upstream_keepalive_module \
    --without-http-cache \
    --without-pcre \
    --without-select_module \
    --without-poll_module
```

Some of these modules could be included in the build, if necessary, but the modules `http_rewrite_module` and `http_gzip_module` require external libraries which are not present on the MicroBlaze system and therefore would not work.

5.3.4 Compiler Configuration

Besides the configuration for *nginx* itself, the compiler needs to be configured for the target *MicroBlaze* system. When building the Linux kernel, the compiler configuration is set by the configuration of the kernel during the build process. *nginx* does not have this functionality in its configuration system, but comes with a parameter (`--with-cc-opt=...`) to pass custom parameters to the compiler.

Using this configuration setting, the GCC cross-compiler needs to be configured for the feature-set of the specific *MicroBlaze* system. The parameters required for the developed *MicroBlaze* SoC are

```
-mxl-multiply-high -mno-xl-soft-mul -mno-xl-soft-div -mxl-barrel-shift ↩
-mxl-pattern-compare -mcpu=v8.30.a
```

Additionally the path to the standard libraries on the system needs to be set through the `--sysroot` parameter. These libraries are part of the tool chain for *MicroBlaze* systems provided by *Xilinx*.

By specifying the `--static` parameter all referenced libraries are build into the resulting binary. Therefore the binary has less dependencies for execution.

Combining it all together, the `--with-cc-opt` parameter needs to be set to the following value:

```
--with-cc-opt="-mxl-multiply-high -mno-xl-soft-mul -mno-xl-soft-div ↩
-mxl-barrel-shift -mxl-pattern-compare -mcpu=v8.30.a --static ↩
--sysroot=/home/peschuster/project/microblaze-unknown-linux-gnu/ ↩
 microblaze-unknown-linux-gnu/sys-root"
```

5.3.5 Memory Leaks

One problem that arose already during very early tests were memory leaks. It turned out that *nginx* allocated about 4,053.2 bytes of memory for each request. Assuming available memory of about 200 MB, *nginx* crashed the complete system after approximately 50,500 requests in total. Obviously this is an unacceptable behavior for a (web) server system.

requests	master / KB	worker / KB
0	3064	3204
1000	3064	7296
2000	3064	11256
3000	3064	15348
4000	3064	19440
5000	3064	23404
6000	3064	27496
7000	3064	31588
8000	3064	35552
9000	3064	39644
10000	3064	43736

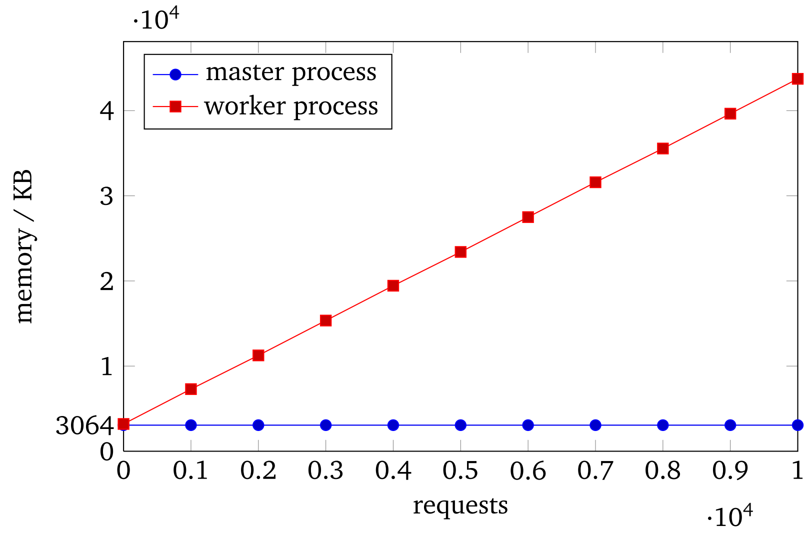

Figure 5.3.: Memory consumption of nginx processes over total requests.

Investigations

The described behavior of *nginx* could not be replicated on a standard x86 server system running *Ubuntu Linux 12.04*. That means *nginx* in general should work correct, using just as much memory as required and releasing unused memory to the OS. But this does not work properly on the *MicroBlaze* system.

With activated `debug` log. *nginx* writes some information about inner workings to the error log. The debug log can be activated with the following option in the *nginx* runtime configuration file:

```
error_log logs/error.log debug;
```

There must be only one `error_log` line in the *nginx* configuration file, but the option specifying the severity level is inclusive for all upper levels.

The following table shows all memory related debug messages as found in the error log for a single request to a static file on the *MicroBlaze* system running an *nginx* instance, which is configured as described in the previous section (sec. 5.3):

Log entry	ptr	allocated	freed
`*1 malloc:  10080890:644`	10080890	644	
`*1 malloc:  10080B18:1024`	10080B18	1024	
`*1 posix_memalign:  1007B5B0:4096 @16`	1007B5B0	4096	
`*1 free:  1007B5B0, unused:  2079`	1007B5B0		4096
`*1 free:  10080890`	10080890		644
`*1 free:  10080B18`	10080B18		1024

Table 5.1.: Memory management related debug messages of a single request.

All pointers to allocated memory for one request are passed to the `free(..)` function of the *C Standard Library* (*libc*) and therefore should be released to the system. But performance and load tests on the system proved a different behavior: The *nginx* worker process consumes more memory for each request with a linear relation to the number of handled requests (see figure and table 5.3).

Therefore it can be assumed that the error causing this misbehavior is not located in *nginx* itself, but in the underlying system layers. This implies that memory management of the Linux kernel or the *libc* port to the *MicroBlaze* architecture are broken in the described respect. While being a strong allegation, especially because fully analyzing the problem on this wide dimensions was beyond the scope of this bachelor thesis, there are a few points promoting this theory:

Xilinx support answer `AR #12421` states that memory management (especially the function `free(..)`) is "very system-specific" and not fully supported and implemented for *MicroBlaze* processors [12]. The answer was published in September 2010 and its validity is explicitly limited to versions of the *MicroBlaze* processors without a hardware memory management unit. Therefore it should not apply to the used version of the *MicroBlaze* processor and *C Standard Library*, but it shows that these memory management functions were added to the toolchain and supporting environment only recently and might not be as stable as other, more major parts.

A possible explanation why the described faulty behavior is only visible when using *nginx*, is the unusual way *nginx* deals with memory. Memory management is done by *nginx*'s pool allocator. This could be seen as an abstraction to the memory allocation mechanisms provided by the OS. When another part of *nginx* requires dynamically allocated memory, it requests it from the pool allocator, which itself requests a larger chunk of memory (called "pool") from the OS and distributes small blocks of the pool on requests from other parts of *nginx*. When free blocks of a pool are exhausted, but none are used anymore, the complete pool is returned to the OS. *nginx* uses this design to minimize system calls and reduce expensive requests for memory allocation by the hardware memory management unit [2]. One consequence of this design is that *nginx* does not reuse once allocated memory, but just allocates new memory blocks when required and releases them to the system upon finished operations. This is by design and beneficial for speed and efficiency, but comes in unfavorable, when memory management of the system (OS and processor) might not work properly.

A First Workaround

It was not possible to fix the root cause of the memory leaks during this bachelor thesis. To circumvent the described arising problems, another solution needed to be found. Otherwise practical usage of the system and extensive performance benchmarks would not be possible.

A workaround to circumvent complete system crashes during tests due to exhausted memory is to restart the *nginx* worker process on low remaining system memory.

This can be accomplished by sending the HUB signal to the *nginx* master process using the kill command of *Unix* systems[5]. The HUB signal tells *nginx* to reload the current configuration, resulting in spinning up new worker processes and gracefully shutting down the previous ones. This differs from complete restarts of *nginx* in the way that no incoming requests are lost during the restart process.[6]

The process id of the master process on the system is always stored in the file /usr/local/nginx/logs/nginx.pid. Therefore the complete command for restarting the *nginx* worker process can be constructed as follows:

```
kill -HUP $( cat /usr/local/nginx/logs/nginx.pid )
```

Information about currently allocated and free memory can be displayed using tools like *top*[7], which provide "a view of process activity in real time" [20]. *top* internally aggregates information from multiple (virtual) file handles like e.g. /proc/meminfo for information about memory.

An Integrated Solution

This workaround is fully functional, but requires manual actions by a user and is therefore ignorant for automation which is a key part of extensive and reproducible performance benchmarks.

An idea for an extended solution was to shift checking for low remaining memory to the *nginx* master process.

After configuring and building up the system, including the start of worker processes, the master process returns to a standby mode, waiting for external signals (like the previously described HUB signal). This is implemented using default *C Standard Library* signal implementations, mainly sigsuspend[8], inside of the ngx_master_process_cycle(..) function (in the file ngx_worker_cycle.c).

A developed patch to check memory consumption of the system and initiate worker process restarts hooks in at this point inside of the master process.

The patch consists of three major parts:

1. Starting a timer, to awake the master process every second from suspension. This is implemented using the Linux command setitimer[9] in ITIMER_REAL mode to send a SIGALRM signal to the current (i.e. *nginx* master) process.

2. A signal handler for dealing with the SIGALRM signal, to check the remaining system memory and trigger a restart of worker processes, if necessary.

3. A function reading the file handle /proc/meminfo, parsing out the required information about free system memory and comparing it with a configured threshold value. This is implemented in the added file ngx_process_memguard.c.

Usage of the newly implemented functionality needs to be activated upon build configuration by providing the parameter --with-min-free-mem=[value]. [value] is the threshold value in kilobytes which is compared to the free system memory to decide on a required restart of worker processes. During tests of the system this value was set to "51200" (kilobytes).

5 http://unixhelp.ed.ac.uk/CGI/man-cgi?kill
6 http://wiki.nginx.org/CommandLine (as of 12/2012)
7 http://www.busybox.net/downloads/BusyBox.html#top
8 http://www.gnu.org/software/libc/manual/html_node/Sigsuspend.html
9 http://linux.about.com/library/cmd/blcmdl2_setitimer.htm

The complete patch is included in the appendix of this report (see B.1.2) or can be found in the master branch of the *nginx* fork at `https://github.com/peschuster/nginx`.

5.3.6 Compilation and Installation

After configuration, the *nginx* binary file is compiled using "`make -f objs/Makefile`". The resulting image file has a size of about 1.8 MB.

Installation into the target directory is initiated executing the command "`make install DESTDIR=<dir>`". "`<dir>`" needs to be replaced with the path to the target directory. For the environment used during this project, this is equal to the directory containing the target root file system "`/home/peschuster/project/customfs/complete`".

Executing the described `install` command creates the required directory structure with default configuration files and copies the binary image file.

5.3.7 Runtime Configuration

Runtime configuration of an *nginx* installation is stored in the `conf/` directory relative to the *nginx* root. For this project and conducted performance tests, there was no need to change the default configuration (`nginx.conf`), which is loaded at application start up when no other options are specified.

5.4 Interactions with the Operating System

This section briefly describes interactions between *nginx* and the underlying OS to deal with HTTP requests and responses.

User applications can interact with the network stack implemented in the Linux kernel using the **socket**[10] interface.

The *nginx* master process opens sockets using the `socket(..)` command. This returns a file descriptor (`int`) which is stored in the `ngx_connection_t` struct and used for subsequent calls to functions of the socket interface. Next the initialized socket is bind to the configured local port using `bind(..)`. This is done for all `listen` directives specified in the configuration file. The default setting "`listen 80`" opens a socket on port `80` for all IP addresses (`0.0.0.0:80`). It is also possible to listen on unix sockets by specifying a configuration value in the format "`listen unix:/tmp/nginx1.sock;`".[11]

Last step during initialization is a call to `listen(..)` to indicate that *nginx* is ready for accepting incoming connections. The `backlog` parameter, indicating the maximum number of incoming connections the OS is allowed to queue up, is set during compile time to 511. This is assumed to be a "safe value" for the Linux OS.[12]

nginx worker processes initialize the core event module, which itself registers the callback `ngx_event_accept(..)` for incoming connections. Inside of `ngx_event_accept` the system call `accept(..)` is made. The result of `accept` calls is again a file descriptor, but just for the single, accepted connection [9]. It is stored in a newly created struct of type `ngx_connection_t` for further references. Information about the remote host is also stored in this new connection struct.

Data is read from sockets using `recv(..)` which is called on parsing the HTTP request header. The call to `recv(..)` simply copies received bytes to the `header_in` field of the current `ngx_request_t` struct.

[10] `http://linux.die.net/man/7/socket`
[11] `http://nginx.org/en/docs/http/ngx_http_core_module.html#listen` (as of 12/2012)
[12] Mailing list answer by Igor Sysoev: `http://forum.nginx.org/read.php?2,9959,9967#msg-9967`

Responses for requests to static files are written to the socket buffer using the two functions `writev(..)` for HTTP header data and `sendfile(..)` for content of the actual static file. `writev(..)` is favored over `write(..)` or `send(..)`, because it allows writing data to a file descriptor from multiple local buffers.[13] *nginx* makes heavy usage of this extended functionality over `write(..)`.

`sendfile(..)` is a function provided by the Linux kernel for copying data between two file descriptors. It is more efficient than a combination of `read(..)` and `write(..)`, because copying is done completely within the kernel without the requirement of "transferring data to and from user space"[14].

An interesting point to note is that the combined use of `writev(..)` and `sendfile(..)` leads always to atleast two TCP packets (see introduction to performance tests: section 6.1.1).

[13] `http://linux.die.net/man/2/writev`

[14] http://www.kernel.org/doc/man-pages/online/pages/man2/sendfile.2.html

6 Performance-Analysis

Before going into the performance tests itself, it is required to understand some fundamentals about network communication and the anatomy of a HTTP request.

6.1 Introduction

"To reduce their design complexity, most networks are organized as a **stack of layers or levels**, each one built upon the one below it" [19][sec. 1.3.1]. The same holds true for the protocol stack driving the "internet". There exist various models for naming and describing the different layers. In context of the internet usually the "TCP/IP Reference Model" is used [19]. This model describes five independent layers, each build upon the next underlying layer.

Real communication between two systems, i.e. exchange of data, always happens on the lowest layer (the "Physical Layer"). But from a logical point of view, communication happens on each layer between the two systems. Therefore each layer might add a header in front of the data, specifying meta information for its counterpart on the other system. Header data of layers above the current layer is treated as if it would belong to the message itself (see figure 6.1).

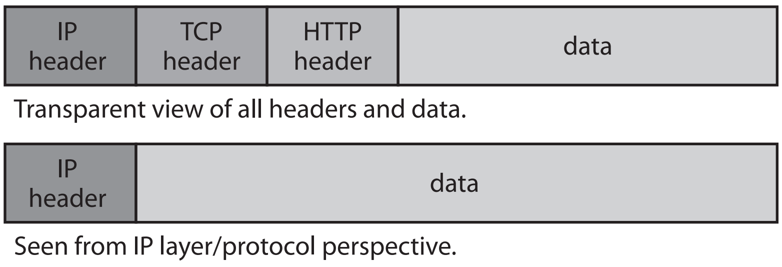
IP header
TCP header
HTTP header
data
Transparent view of all headers and data.
IP header
data
Seen from IP layer/protocol perspective.

Figure 6.1.: Header data for multiple protocol layers (3-5).

Figure 6.2 shows the five layers of the TCP/IP model with corresponding communication protocols and modules implementing the protocol on the developed test system. On construction of single parameters for a performance benchmark, it needs to be taken into account which parts of the system are effected by this parameter and might influence the results.

Layers	Protocol	Implementation
5 - Application	HTTP	nginx
4 - Transport	TCP	Linux kernel
3 - Network	IP	
2 - Data Link	Ethernet	
1 - Physical	PHY	Hardware

Figure 6.2.: Layers of the network stack with reference to implementing modules.

6.1.1 Maximal Transfer Unit (MTU)

Maximal Transfer Unit (MTU) is a parameter specifying the maximum size of a single unit transferred over the network in bytes. It is set for a complete network (sender, receiver, intermediate stations), specifying and limiting its capacity to a robust and reliable usage level. When MTU values for connected stations differ, the lowest takes effect.

In Ethernet networks the MTU is usually set to 1500 bytes [19]. The MTU value as used in Linux includes only message data as received by the Ethernet layer. Header data of the Ethernet protocol does not count into this size limit. When messages (including header data of upper layers) exceed the MTU size, they are split into multiple packets. In practice this means that for transmitting a single HTTP message (i.e. request or response) two or more TCP messages might be required.

6.1.2 Transmission Control Protocol (TCP)

Although HTTP is not fixed to be used only with TCP; HTTP on top of TCP on top of IP is the most dominant usage combination. This is, because TCP provides reliable communication on top of unreliable networks [16]. TCP accomplishes this by setting up and maintaining a connection. Therefore sending data over TCP consists of three steps: establishing a connection, sending the actual message, closing the connection. To guarantee arrival of sent data, TCP works with a handshake protocol between both involved parties for connection state management and acknowledge messages for received packets.

The overhead of maintaining a connection is the price to be paid for a reliable communication channel between two systems. But it also allows sending more than one message using an already established connection. This comes in handy, when the size of a message exceeds the MTU or multiple HTTP requests are send to a server (used in persistent HTTP connections, also called "HTTP keep-alive" [17]).

The *Internet Protocol* (IP) introduced so called IP addresses as identifiers for single systems. TCP extends this model by port numbers. An application providing services to other systems (i.e. a server) uses one single port, but client applications need to open separate ports for each TCP connection. This is required, because TCP connections are identified by the four parts "server address", "server port", "client address" and "client port" [16]. The TCP header contains two 16 bit fields for source and destination port [19]. This leads to a maximum of $2^{16} = 65{,}536$ concurrently used (open) TCP connections on a single client system. This limitation on the capacity of clients needs to be taken into consideration for the design of performance benchmarks. Otherwise observed performance limits might be mistakenly interpreted as limits of the server system [6].

6.2 Measuring Web Server Performance

Pure network performance is often measured as throughput in bits per second. While this gives a great insight in the capabilities of the network, there is no indication as to how many users can be served concurrently by a web server running on the system under test. To gain insights in this performance parameter, dedicated tests on HTTP level need to be performed.

A simple method for measuring performance in terms of request throughput is to "send requests to the server at a fixed rate and to measure the rate at which replies arrive" [6]. By monotonically increasing the request rate, the server becomes saturated at a certain point. This becomes visible in leveling off reply rates and shows that the server operates at full capacity. [6]

6.2.1 Tools

httperf

httperf[1] is a tool developed by David Mosberger and Tai Jin at Hewlett-Packard to specifically measure web server performance [6].

Characteristics of performed tests can be specified by command line arguments to *httperf*. Besides a number of self-explanatory parameters, there are three relevant parameters shaping the actual performance test: `rate`, `num-conns` and `num-calls`.

The `rate` parameter specifies the rate at which connections are created per second. `num-conns` specifies the total number of connections to be created during one test run. `num-calls` specifies the number of requests made over a single TCP connection before it is closed.

A call to *httperf* used for performance tests looks like the following:

```
httperf --timeout=5 --client=0/1 --server=192.168.2.125 --port=80 --uri=/index.html ↩
--rate=120 --send-buffer=4096 --recv-buffer=16384 --num-conns=3000 --num-calls=3
```

When executing *httperf* a warning might be displayed:

```
warning:  open file limit > FD_SETSIZE; limiting max.  # of open files to FD_SETSIZE
```

What it basically means is that the process reached the limit of concurrently allowed open file descriptors. This is a per process limit enforced by the OS. *httperf* is using the *select*[2] module for dealing with network tasks and creates a new file descriptor for every TCP connection. Therefore the limit on open file descriptors limits the number of usable TCP connections by *httperf* and might influence the result of performance tests.

[1] `http://www.hpl.hp.com/research/linux/httperf/`
[2] "`select`" is an event library provided by the Linux kernel.

To fix this issue, limits on the current system need to be increased and *httperf* re-compiled. This includes three steps[3]:

1. To increase the hard limit of open file descriptors for all users, the following line needs to be inserted (or updated) in `/etc/security/limits.conf`: "`* hard nofile 65535`"

2. The value of the pre-compiler constant "`__FD_SET_SIZE`" needs to be increased to `65535`. It is defined in "`/usr/include/bits/typesizes.h`".
   This file might be located in another sub-directory, e.g. "`/usr/include/x86_64-linux-gnu/bits/typesizes.h`".

3. *httperf* needs to be re-compiled from source:

   ```
   wget ftp://ftp.hpl.hp.com/pub/httperf/httperf-0.9.0.tar.gz
   tar xvzf httperf-0.9.0.tar.gz && cd httperf-0.9.0
   ./configure && make
   sudo make install
   ```

A useful helper for automating test runs with increasing connection rate is *autobench*[4]. It is a Perl script acting as a wrapper around *httperf* and was published by Julian Midgley.

Custom Scripts

In addition to accurate engineered test cases using *httperf*, custom scripts were used to generate load on the server. A small Windows console application written in `C#`, utilizing the *Task Parallel Library*[5], turned out to be the most efficient one with best performance characteristics.

This console application is also included in the appendix of this report (see B.2.2).

Wireshark

Wireshark[6] is a network protocol analysis software, which captures all traffic going through a network interface. It especially helps in analyzing and inspecting captured data through its deep knowledge of all common protocols.

It was used in this project for debugging initial problems with the network stack and low-level analysis of performance test parameters.

6.3 Tests

The tests were conducted using *httperf* on a virtual machine running *Ubuntu Linux 12.04*. To proof that the limiting factor during the tests is not this client machine, but the system under test, all tests were also performed against a *Microsoft IIS 7.5* web server instance, which was hosted on a bare metal machine with an *AMD Phenom II X4 965* processor at 3.40 GHz, running *Microsoft Windows 7 Professional* as operating system.

According to statistics of the *HTTP Archive*[7] an average HTML page has a size of about 6 KB. Other content types, like images, stylesheets and script files vary on average between 5 KB and 21 KB.

[3] The described solution to the FD_SETSIZE problem was taken from Guillaume Maury: `http://gom-jabbar.org/articles/2009/02/04/httperf-and-file-descriptors`.
[4] `http://www.xenoclast.org/autobench/`
[5] `http://msdn.microsoft.com/de-de/library/vstudio/dd460717.aspx`
[6] `http://www.wireshark.org`
[7] `http://httparchive.org/interesting.php#responsesizes` (as of Dec. 15th 2012)

Taking this usage pattern of websites into consideration, tests were performed using static files with two sizes: one file having exactly 10 KB (10.240 bytes) from now on referred to as "10K.html" and one very minimal HTML page having just 354 bytes ("index.html"). The major difference between these two files from a "network perspective" is that the small file can be transmitted in two TCP packets, one is used for the HTTP header and one for the actual data (see section 5.4), whereas the 10 KB file needs to be split up in 9 TCP packets due to the default MTU size of 1500 bytes. A third test-run was performed requesting a file that does not exist. This results in a HTTP response with code "404 – Not Found". It is a special case, because the response is transmitted in a single TCP packet and no file needs to be read from the file system.

Figure 6.3 shows request and response rates for all three test cases with an increasing connection rate (20 conn./s to 120 conn./s, step size: 10). On each connection three HTTP requests were made to the respective file.

On every connection rate level 3,000 connections were opened to obtain reliable results. Therefore each test run included 33,000 connections and up to 99,000 requests.

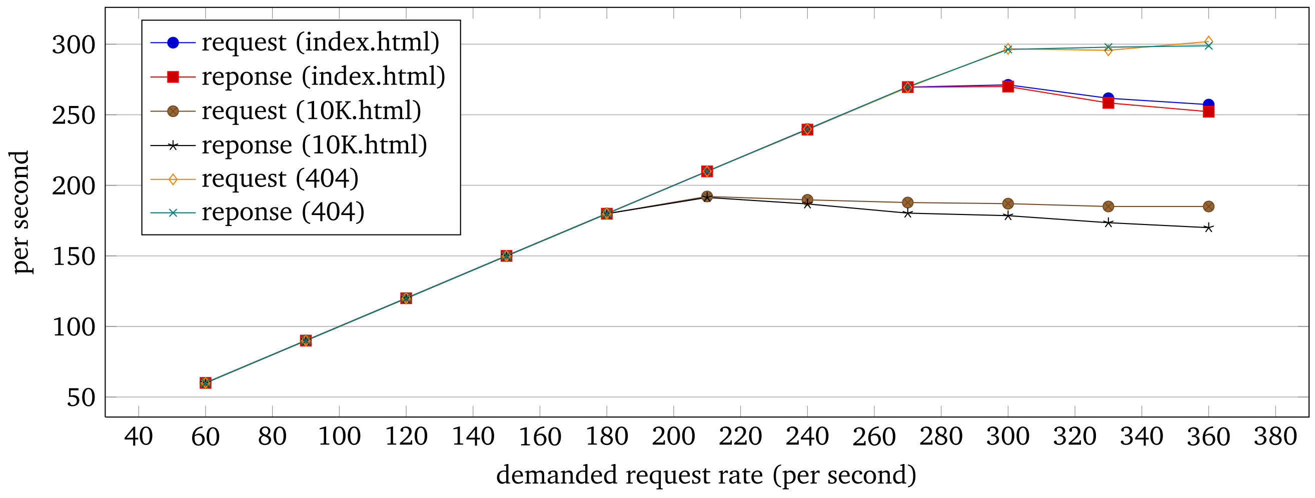

Figure 6.3.: Request and response rates for the different test runs.

The test result shows that at a certain request rate, the response rates levels off. For the two test runs requesting files, response rates even decrease slightly. At this points the server becomes saturated and the number of requests oversteps its actual capacity. These points are namely at about 300 req./s for the very small response, at about 270 req./s for the "index.html" file and at about 190 req./s for the "10K.html" file.

The difference between request and response rate origins in failed requests. The following figure shows the corresponding error rate. It increases with over-saturating the system.

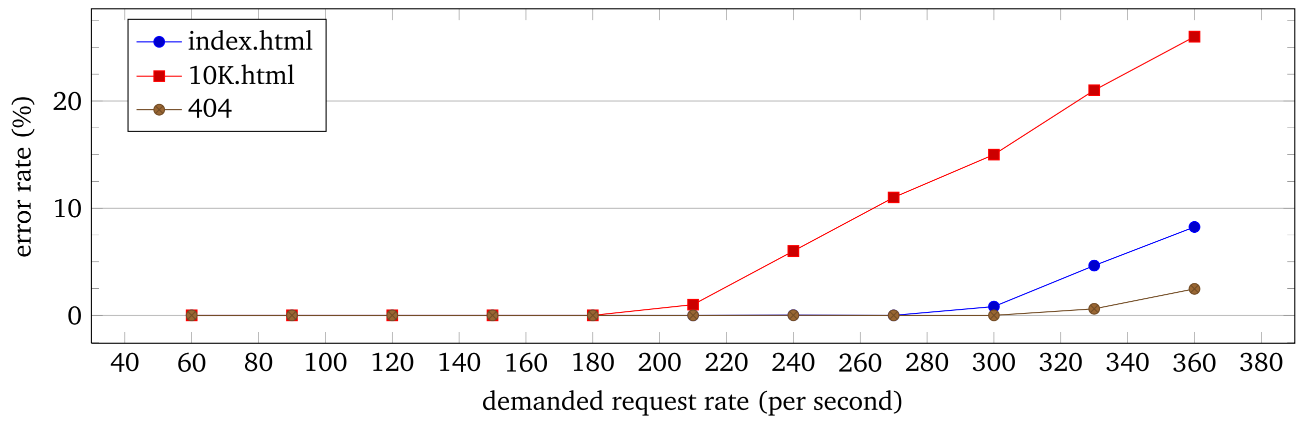

Figure 6.4.: Error rate (failed requests) for the different test runs.

The response **rate** decreases only slowly with growing saturation, but the average response **time** increases from 5 ms to about 500-700 ms:

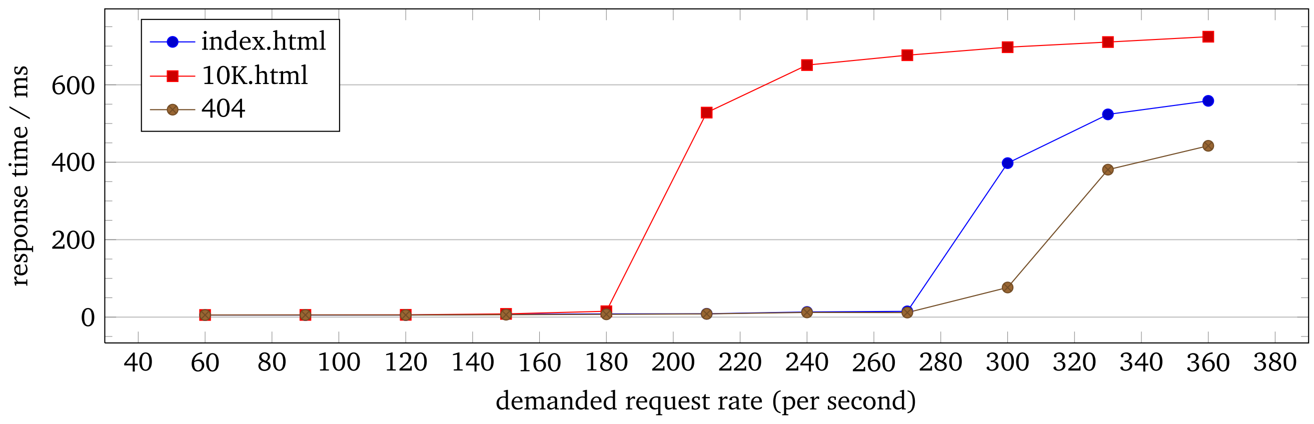

Figure 6.5.: Response time for the different test runs.

The reason for decreasing slopes in response times might be a clogged backlog for incoming connections, resulting in an early, and therefore cheap rejection of further incoming requests.

The difference in request and response rates the system can handle is also visible in **CPU utilization**. Main responsibility of *nginx* for serving requests is to handle the HTTP part of the process. This includes parsing request headers, generating responses and delivering static files. The OS on the other hand does the "translation" to and from TCP and handles all work on lower network layers (see also figure 6.2). An increased payload size of single responses entails therefore a shift in processing time from *nginx* ("user space") towards the operations inside the Linux kernel. Figure 6.6 shows exactly this correlation.

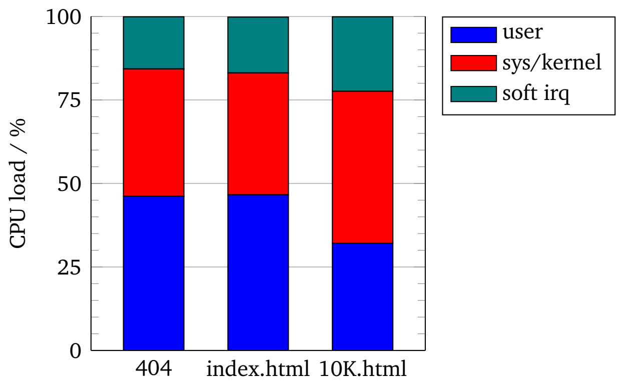

Figure 6.6.: CPU load distribution for different request types.

Network **throughput** of the tests follows naturally the trend of response rates. For tests requesting the "index.html" file and those resulting in "HTTP 404 – Not Found" responses, the shown throughput is very low, because *httperf* calculates throughput using the formula $\frac{[\text{size}]\cdot[\text{total repsonses}]}{[\text{test time}]}$. Therefore very small response sizes yield a low average throughput.

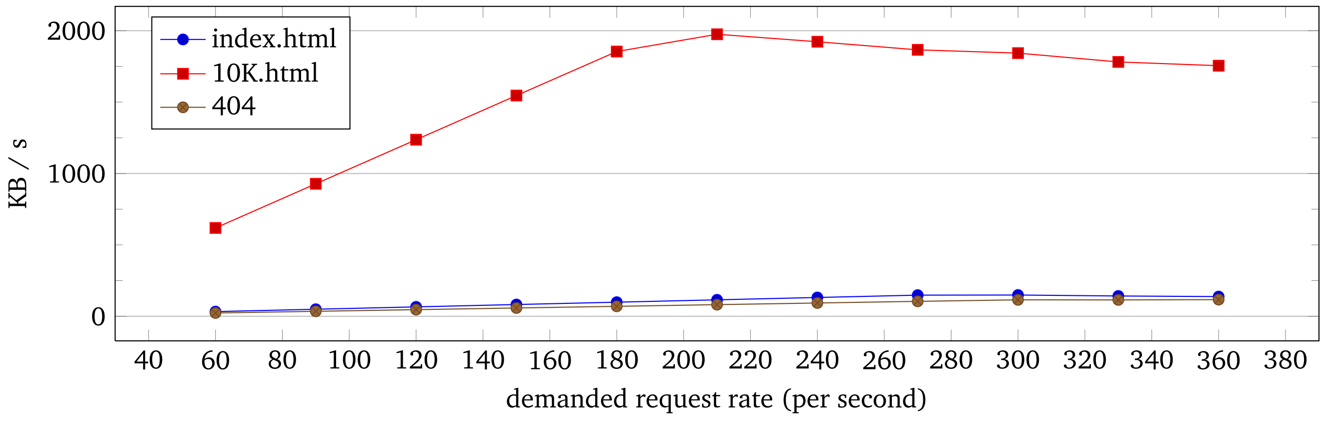

Figure 6.7.: Throughput for the different test runs.

6.3.1 Dedicated Network Throughput Tests

To test the maximal network throughput the server can provide, a test needs to be designed focusing on just this parameter. This approach prevents other parts of the system from getting into the way. A constant high exchange of TCP packets can be achieved by transferring very few, but large files over the network. Therefore a large binary file having a size of 33.4 MB was used for throughput tests.

Figure 6.8 shows the network throughput for requests to this file with an increasing connection rate and two HTTP requests on each connection. The system under test goes into saturation at about 61 Megabits/sec. This is equivalent to 6.1 % of the available network bandwidth, providing a maximum of 1 Gigabit/sec. It is important to note that the number of 61 Megabits/sec. is not equivalent to the number of actually transferred bits over the network, but only includes the received "usable" data. Transferred connection setup and close, acknowledgment and TCP, IP and Ethernet header messages are not included in this number.

The figure also shows the total CPU utilization of the tested system.

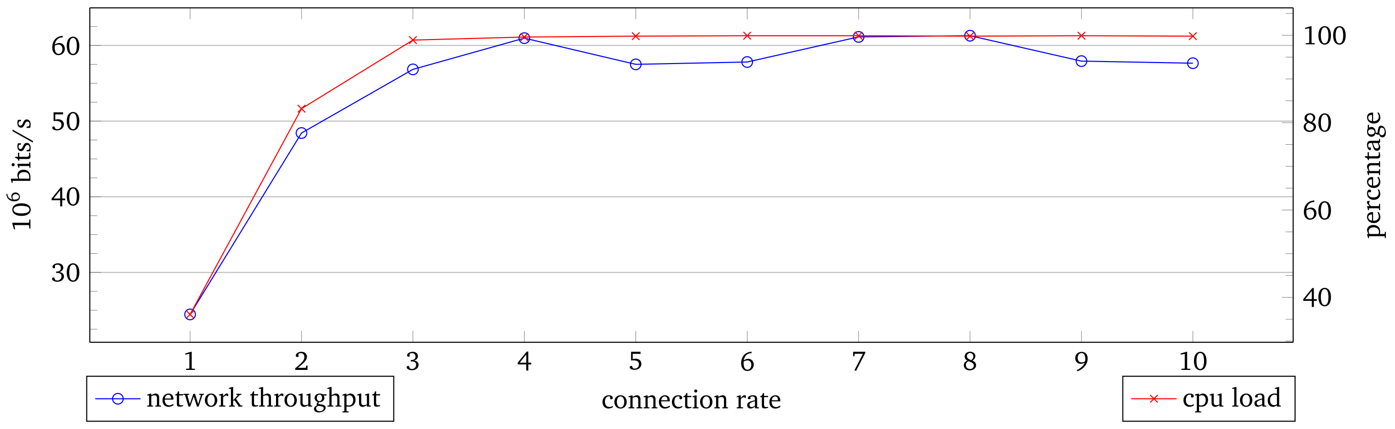

Figure 6.8.: Network throughput and CPU load.

Figure 6.9 shows the CPU utilization by time spent in the different types of the system. It shows that most time is spent in kernel space and handling interrupt requests by the network driver and sub system (soft irq). nginx (user space) takes only very low CPU utilization and is not the limiting factor in this test. This conforms with test results of the previous tests on smaller files.

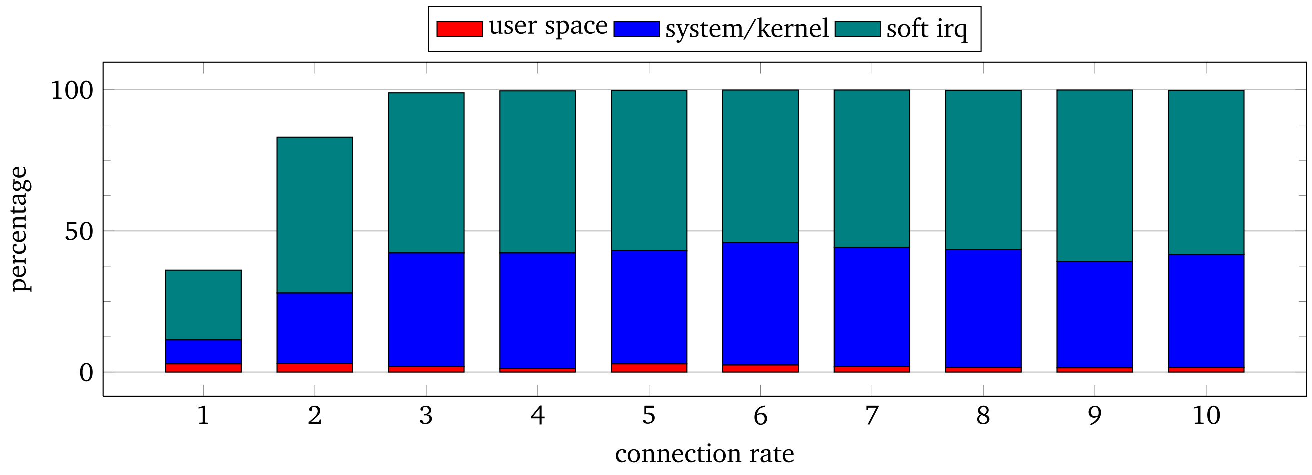

Figure 6.9.: CPU utilization by category.

To proof that obtained performance limits were not limits of the client executing the tests or the network infrastructure, the same test was also executed against a "reference system" running a *Microsoft IIS 7.5* web server. Figure 6.10 shows the results. Measured throughput (averaged per request) had a maximum at about 515.7 Megabits/sec which outplays the results of the *MicroBlaze* system by more than a factor of eight.

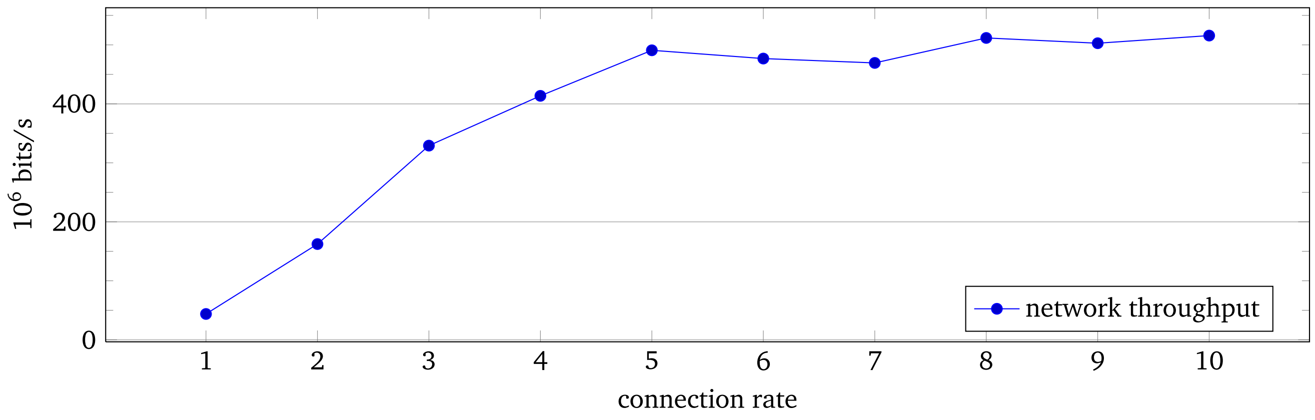

Figure 6.10.: Network throughput of an AMD Phenom II X4 processor at 3.40 GHz running Microsoft IIS 7.5 on Windows 7.

6.4 Discussion of Results

Obtained performance results show that the designed System on Chip with custom configured and compiled Linux kernel is capable of running a web server serving up to about 300 requests per second and 100 concurrent connections. Compared to the low performance indicator of 48.94 BogoMIPS[8], which is less than a $\frac{1}{130}$ of the value Linux kernel reports for a virtualized *AMD Phenom II X4* processor at 3.40 GHz running on two cores[9], this is not to bad. But the capacity of the server would not be sufficient for running it as a public web server connected to

8 "The number of million times per second a processor can do absolutely nothing." (untraceable source: Eric S Raymond, Geoff Mackenzie – `http://www.tldp.org/HOWTO/html_single/BogoMips/`)

9 6414.98 BogoMips

the global internet infrastructure. Especially because response rates rapidly decrease with increasing byte size of responses, reasoned in a larger amount of TCP packets required for transmitting the response.

A reduction in the number of TCP packets could be achieved by using so called "Jumbo Frames". These are Ethernet frames with a data size larger than the default MTU of 1500 bytes. The responsible setting can be changed executing the command "`ifconfig eth0 mtu [new size]`". Where `[new size]` is the new MTU size in bytes. Unfortunately this does not seem to be fully supported by the *Xilinx LL TEMAC* Ethernet driver[10], leading to TCP packets with a total size larger than 1500 bytes to be corrupted and therefore lost "on the network". Besides this, Jumbo Frames would have to be supported on the client and all intermediate stations as well, which can not be guaranteed on a public network or on providing a public service.

nginx provides a configuration setting "`worker_connections`" which limits the maximum amount of concurrently handled connections by a worker process. During tests this setting had a value of 768. But neither changing the number of maximum allowed connection to 1024 nor 512 had any influence in observed response rates. Therefore it is safe to be assumed that no internal limit for concurrent connections affected the test results. But therefore this part does not have any room for improvements, either.

Especially the tests for maximum network throughput showed that the main bottleneck is in the TCP and network stack on the test system. Besides providing more processing power to the system, optimizing this part of the Linux kernel is hardly possible in the project context. Another option would be to offload processing of TCP connections, requests and responses to dedicated hardware, implemented as a network interface with enhanced functionality. This possible solution is also refereed to by the title of this thesis using the word "accelerated", but requires deep integration into already available functions of the operating system. Therefore the next two chapters will provide an overview of the Linux network stack and possible integration options for a *TCP Offload Engine* (TOE).

[10] see Xilinx forum: `http://forums.xilinx.com/t5/Embedded-Linux/Jumbo-frames-ifconfig-crash-and-xps-ll-temac-issues/td-p/211911` (as of 12/2012)

7 Overview of the Linux Network Stack

The top most layer of the Linux network stack is the **system call interface** to create and operate on sockets [14]. The usage of this interface by *nginx* is already described in section 5.4. Its purpose is to multiplex networking calls by the user into the kernel [14]. Once a file descriptor was created using the socket interface, data can be exchanged with the network system through general operations on file descriptors like *write* and *read*, too. The socket interface is completely agnostic to different protocols or implementations by network devices and drivers. Its main underlying structure is "`struct sock`", containing all relevant information about a specific socket, like available functions and protocol specific state information.[1] Thereby also protocol and implementation specific functionality is bind to a socket using function pointers (defined in "`struct inet_protosw`") [14].

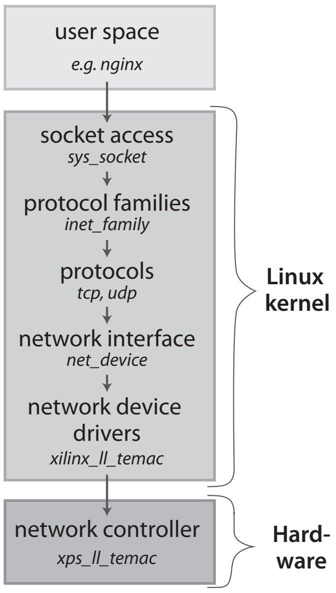

Figure 7.1.: Linux network stack.

Whereas the more general data structure "`struct sock`" contains mainly meta information about a socket, the major structure for storing data is "`struct sk_buff`". **`sk_buff`** contains packet data and state information. It is used for to be sent, as well as for received packets almost through out all layers of the network stack. [14]

The gap between protocol handling and device drivers is bridged by the **network interface** (*netif*) layer. This is a hardware device "agnostic interface layer" [14]. Its major purpose is to connect protocols to hardware devices. Information about devices is provided through "`struct net_device`". On system start-up all available devices register themselves with a filled out "`net_device`" structure. From there on they are know to the network interface layer.

The lowest layer of the network stack being part of the Linux kernel is formed by **device drivers**. These hook into the network interface layer and manage physical/hardware network controllers. For the implemented System on Chip this is the `xps_ll_temac` IP core.

Since introduction of the "New API" (*NAPI*) for communication between device drivers processing packets and the network interface layer, most of the workload for received and to be sent packets is scheduled using software interrupt requests (also known as "*soft irq*" or "*sirq*").[2] During performance tests described in the previous chapter this was visible through high CPU utilization of the *sirq* category.

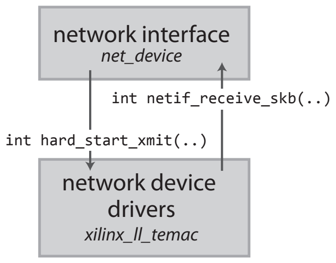

Figure 7.2.: `sk_buff` exchange

The network interface layer enqueues `sk_buff` packets for transmission using the function `int hard_start_xmit(struct sk_buff *skb, struct net_device *dev)`. Usually a call to this function is protected with a *lock* to protect it from being called multiple times simultaneously.[3] The device driver pushes received packets to the upper layer using `int netif_receive_skb(struct sk_buff *skb)`. [14]

[1] `http://www.ecsl.cs.sunysb.edu/elibrary/linux/network/LinuxKernel.pdf`
[2] `http://www.linuxfoundation.org/collaborate/workgroups/networking/napi` (as of 12/2012)
[3] `http://lwn.net/Articles/120960/` (as of 12/2012)

8 TCP Offload Engine

8.1 Overview

8.1.1 Benefits

Conducted performance tests showed[1] that much processing time is spend handling interrupt requests of the network sub system. Especially on TCP-heavy-requests this circumstance tuned out to be a bottleneck for a system with little processing power like the designed System on Chip.

Handling of TCP connections and processing messages is on the other hand a well defined and general problem that could be outsourced to an own "processor", leaving more processing power of the main system (CPU) for other tasks. This is the approach of *TCP Offload Engines* (TOE).

8.1.2 State of the Art

Throughout recent years some attempts by hardware vendors developingBücher network interface cards were made to integrate support for *TCP Offload Engines* into the Linux kernel (e.g. by *Chelsio Communications*: `http://lwn.net/Articles/147289/`). However, maintainers[2] of the Linux network stack and the *Linux Foundation*[3] have a strong opinion against integrating drivers and support through out the network stack for these hardware devices.

This is partially based on reservations about quality standards and the lack of possible benefits, but includes also arguments of a more general nature. A *TOE* would "short out much of the Linux networking code" and therefore "cut out little features like *netfilter*, traffic control, and more" [3]. Additionally it is not easily possible to supply security fixes for *TOE* functionality implemented in hardware. Therefore it would be necessary to cut off integration of a *TOE* on occurring security vulnerabilities by releasing Linux kernel hot fix versions.

These are reasons why no support for *TCP Offload Engines* is available in Linux kernel, currently. Of course all arguments are based on the point of view of Linux being a general purpose operating system used in desktop and server systems, powered by decent computer hardware. For embedded systems these assumptions are usually not fulfilled. Neither will the *TOE* be part of an exchangeable and replaceable network interface card, supplied by a third-party vendor, but probably integrated into a System on Chip as an IP core. Therefore the next sections will discuss approaches for an integration of dedicated engines, offloading TCP processing to hardware.

8.2 Possible Approaches

There are a number of levels for offloading features to hardware. The simplest one is offloading checksum calculation to the network interface card. This is already implemented by the used *xps_ll_temac* IP core (see sec. 3.1.1).

8.2.1 Large Segment and Receive Offload (LSO/LRO)

Another option which is already supported by the network device layer of the Linux kernel is *TCP Segmentation Offload* (TSO), also known as *Large Segment Offload* (LSO).[4] Inside of the Linux kernel, this technique uses very

[1] see figures 6.6 and 6.9
[2] `http://lwn.net/Articles/148701/`
[3] `http://www.linuxfoundation.org/collaborate/workgroups/networking/toe`
[4] `http://www.linuxfoundation.org/collaborate/workgroups/networking/tso` (as of 12/2012)

large data buffers in a single `sk_buff` structure and therefore large TCP packets far beyond typical MTU values. Splitting these single packets to multiple packets of transmittable size (i.e. segmentation) is done by the network device, connected to the Linux kernel as a hardware device. This reduces the number of processed TCP packets and acknowledgments by the CPU, but leaves all state related tasks to the operating system. Therefore it prevents short cutting the network stack and additional functionality for network management and analysis. But it implicates known problems of transparent segmentation, too [19][sec. "5.5.7 Segmentation"].

The other way round is called *Large Receive Offload* (LRO). This concept is supported by the *New API* (NAPI) in Linux kernel for receiving network packets in the way that the number of hardware interrupt requests is reduced on high network utilization, but implemented completely in software [4].

TCP Chimney Offload

Microsoft has introduced a technique called *TCP Chimney Offload* with *Microsoft® Windows Server® 2003 Scalable Networking Pack (SNP)*. This allows offloading the complete handling of TCP processing "on demand" to the network interface card. Connections are therefore known to the operating system and can be managed complete in software. But it is also possible to offload further processing of a connection to hardware to alleviate bottlenecks. Because not only packet, but also connection and state handling can be done by the hardware, the network interface card requires a complete implementation of the TCP/IP protocol stack. [15]

This is a proprietary solution by Microsoft and some hardware vendors. There are no approaches of bringing it to the Linux kernel or other unix-based operating systems, to the knowledge of the author.

Full-stack TCP Offload

The last to be introduced solution can be described as "*Full-stack TCP Offload*". All processing of TCP and IP concerns is done by hardware. This requires a great deal of protocol knowledge and functionality to be implemented in hardware, but promises also the best performance gains, leaving just a minimal part of TCP/IP stack processing to the operating system.

This is the chosen way by the research project enclosing this thesis. Therefore the next section provides some considerations for integrating a *TOE* following this concept into the Linux kernel and remarks concerning device driver development and interfaces of the *TOE*.

8.3 Integration into Linux

Integration of a *Full-stack TOE* into the network stack of Linux kernel must consist of two major parts:

1. A network device driver handling communication with the actual *TCP Offload Engine* device.
2. Changes to the current network stack for – conditionally – short cutting the existing TCP processing implementation.

Chelsio Communications, a hardware vendor, developing network interfaces, made an attempt to integrate an own TOE following this approach into the Linux kernel in 2005.[5] Their solution called "*OPEN TOE*" claimed to be vendor neutral and could therefore serve as a model for a custom implementation.[6]

[5] `http://lwn.net/Articles/147289/` (as of 12/2012)
[6] `http://lwn.net/Articles/146060/` (as of 12/2012)

Central unit for communication with the TOE device is the new `struct toedev`. It represents "a new type of extended network device [. . .] with an additional set of methods" [3].

The provided additional methods are listed below:

```
int (*open)(struct toedev *dev);
int (*close)(struct toedev *dev);
int (*can_offload)(struct toedev *dev, struct sock *sk);
int (*connect)(struct toedev *dev, struct sock *sk);
int (*send)(struct toedev *dev, struct sk_buff *skb);
int (*recv)(struct toedev *dev, struct sk_buff **skb, int n);
int (*ctl)(struct toedev *dev, unsigned int req, void *data);
void (*neigh_update)(struct net_device *lldev,
     struct toedev *dev,
     struct neighbour *neigh, int fl);

Source: http://lwn.net/Articles/147289/
```

The methods can easily mapped to known features of TCP:

`open(..)`, respectively `connect(..)` for client scenarios, signals the availability for incoming TCP connections or initiates the creation of an outgoing TCP connection. `close(..)` ends an existing TCP connection.

The two functions `send(..)` and `recv(..)` are well known from the socket interface layer and facilitate transmission and receiving of single or multiple TCP packets in form of `sk_buff` structures.

`can_offload(..)` and `ctl(..)` provide control access to the TOE device. The function `neigh_update(..)` is required for usage of the *neighbor subsystem*[7] which enables discovery and mapping between physical (*MAC*) and logical (*IP*) network addresses.

The actual data in form of `sk_buff` structures should be transferred to and from the TOE device using DMA. DMA stands for *Direct Memory Access* and describes an architecture featuring direct memory access of devices to system's main memory. Therefore no processing power is deducted from the CPU shifting data between the Linux kernel structures and the TOE device.

One challenge of DMA usage combined with a MMU is the translation between virtual (used by the OS) and physical (used by the device) memory addresses. Continuous memory regions in virtual address space do not need to be continuous in physical address space. Therefore the device might be required to access multiple fragments of memory, although it was passed only a single data structure written by the Linux kernel.

DMA is on the other hand widely used for high bandwidth devices, which is why many references for these problems should exist, alleviating the implementation of appropriate solutions.

Without further knowledge about the concrete TOE implementation, this is as far as preparations and research on the topic of a possible integration into the Linux kernel can go. Next steps would be to work on that implementation.

[7] `http://www.linuxfoundation.org/collaborate/workgroups/networking/neighboringsubsystem` (as of 12/2012)

9 Conclusion

9.1 Project Review

I chose this project, because it promised to cover many topics of personal interest, including network systems, performance oriented research and optimization, web technology and *System on Chip* design. In retrospective these expectations were fulfilled, but came with a steep learning curve and many unexpected challenges.

Work on the bachelor thesis covered the second part of a larger project. Therefore a deep knowledge of the general topic was already available and it was possible to work on the project objectives right from the start.

Practical implementations always come with a great deal of uncertainty regarding occurring problems. In this part of the project, these turned out to be initial incompatibilities between compiler, *Linux kernel* source and *nginx* source versions. As well as memory leaks in *nginx*'s request processing, which seemed to stay unresolvable for a major part of the project time.

One thing that turned out very valuable, was the consequent application of automation, where ever possible. This lead to hassle-free implement-build-deploy-cycles facilitating quick changes with great impact also in a late stage of the project.

Original objective of this project and main reason I chose this as a combined project seminar and bachelor thesis, was the integration of a *TCP Offload Engine* into a System on Chip and Linux network stack. When it turned out that the *TCP Offload Engine* project, being part of a dissertation, will not reach the necessary majority level to be integrated into another system in time, this thesis turned into a pure software project. Making the shift from the original objectives to purely preparatory work for a possible integration, was one of the most challenging aspects, from a motivation perspective.

In review I personally would assess the project as still being successful. Mainly because of all the accomplished integration work up to the reliable running *nginx* instance and all the insights gained on the inner workings of Linux kernel and its network stack implementation.

Due to external circumstances, the original objective of the project was not reached. Therefore an obvious next step would be to finally integrate the *TCP Offload Engine* into a Linux kernel build running on a custom *System on Chip*. Besides this there are always possibilities for optimization of previously implemented features, but without integration of the *TCP Offload Engine*, these probably would not yield a great gain in the overall system performance.

Acronyms

AMBA Advanced Microcontroller Bus Architecture. 5

AXI Advanced eXtensible Interface. 5, 6

CPU Central Processing Unit. 3, 30, 32

DMA Direct Memory Access. 6, 32

DoS Denial of Service. 3

ELF Executable and Linkable Format. 8

FPGA Field Programmable Gate Array. 3–6, 8, 9

GCC GNU Compiler Collection. 7, 9, 14

GMII Gigabit Media Independent Interface. 6

HTTP Hypertext Transfer Protocol. 11, 12, 20–22

IP Intelectual Property. 3, 5, 6, 21, 29–31

MAC Medium Access Control. 3, 6

MII Media Independent Interface. 6

MMU Memory Management Unit. 4, 5, 32

MTU Maximal Transfer Unit. 21, 24, 28, 31

NFS Network File System. 9, 10

OS Operating System. 4, 7, 9, 11–13, 15, 16, 18, 22, 25, 32

PLB Processor Local Bus. 5, 6

PLL phase-locked loop. 6

PVR Processor Version Register. 7

SoC System on chip. 5, 6, 8, 13, 14

TCL Tool command language. 7

TCP Transmission Control Protocol. 11, 21, 25, 31, 32

TOE TCP Offload Engine. 11, 31, 32

XMD Xilinx Microprocessor Debug Engine. 8, 9

XPS Xilinx Platform Studio. 3, 7

Bibliography

[1] 802.3-2000 - part 3: Carrier sense multiple access with collision detect on (csma/cd) access method and physical layer specifications. *IEEE Std 802.3, 2000 Edition*, pages i –1515, 2000.

[2] Andre Alexeev. *The Architecture of Open Source Applications, Volume 2*, chapter nginx. Amy Brown, Greg Wilson, 2012.

[3] Jonathan Corbet. *Linux and TCP offload engines*. Linux Weekly News, `http://lwn.net/Articles/148697/`, August 2005.

[4] Jonathan Corbet. *Large receive offload*. Linux Weekly News, `http://lwn.net/Articles/243949/`, August 2007.

[5] Intel Corporation. *Endianness White Paper*. `http://www.intel.com/design/intarch/papers/endian.pdf`, 11 2004.

[6] Tai Jin David Mosberger. *httperf – A Tool for Measuring Web Server Performance*. Hewlett-Packard Co., `http://www.hpl.hp.com/research/linux/httperf/wisp98/httperf.pdf`, December 1998.

[7] Bruce Fienberg. *Xilinx Acquires Embedded Linux Solutions Provider PetaLogix*. `http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1729100`, 8 2012.

[8] David Gibson and Benjamin Herrenschmidt. *Device trees everywhere*. `http://ozlabs.org/~dgibson/papers/dtc-paper.pdf`, 2 2006.

[9] Brian Hall. *Beej's Guide to Network Programming*. Jorgensen Publishing, 2011.

[10] Christopher Hallinan. *Embedded Linux Primer: A Practical Real-World Approach*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[11] Klaus Mochalski Hendrik Schulze. *ipoque GmbH*. `http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf`, 2009.

[12] Xilinx Inc. *MicroBlaze - How is memory managed in MicroBlaze? Should I use malloc/free functions to manage memory?* `http://www.xilinx.com/support/answers/12421.htm`, 2010.

[13] National Instruments. *FPGA Fundamentals*. `http://www.ni.com/white-paper/6983/en`. called 08/28/2012.

[14] M. Tim Jones. *Anatomy of the Linux networking stack*. Emulex Corp., `http://www.ibm.com/developerworks/linux/library/l-linux-networking-stack/`, June 2007.

[15] Ian Hameroff Pankja Gupta, Allen Light. *Boosting Data Transfer with TCP Offload Engine Technology*. Dell Inc., `http://www.dell.com/downloads/global/power/ps3q06-20060132-Broadcom.pdf`, August 2006.

[16] Jon Postel. *Transmission Control Protocol (RFC 793)*. Information Sciences Institute, University of Southern California, 1981.

[17] J. Mogul H. Frystyk R. Fielding, J. Gettys and T. Berners-Lee. *Hypertext Transfer Protocol - HTTP/1.1. (RFC 2612)*. Internet Engineering Task Force, June 1999.

[18] Peter Schuster. *Design of an Accelerated Event-based Server*. http://www.peschuster.de/ies-project/ProjectReport-Http.pdf, 10 2012.

[19] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[20] Denis Vlasenko et al. *BusyBox - The Swiss Army Knife of Embedded Linux*. http://www.busybox.net/downloads/BusyBox.html.

[21] Xilinx. *XUPV5-LX110T User Manual*. http://www.xilinx.com/univ/xupv5-lx110T-manual.htm. called 09/06/2012.

[22] Xilinx. *Virtex-5 Family Overview (DS100)*. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, v5.0 edition, 2 2009.

[23] Xilinx. *LogiCORE IP XPS LL TEMAC*. http://www.xilinx.com/support/documentation/ip_documentation/xps_ll_temac.pdf, v2.03a edition, 12 2010.

[24] Xilinx. *ML505/ML506/ML507 Evaluation Platform - User Guid (UG347)*. http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf, v3.1.2 edition, 5 2011.

[25] Xilinx. *LogiCORE IP AXI Interconnect*. http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v1_06_a/ds768_axi_interconnect.pdf, v1.06a edition, 4 2012.

[26] Xilinx. *MicroBlaze Processor Reference Guide (UG081)*. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf, 14.1 edition, 4 2012.

[27] Xilinx et al. *MicroBlaze Linux (General)*. http://wiki.xilinx.com/microblaze-linux.

A Performance Test Results

Results of the first set of tests

Demanded Req. Rate (1/s)	**Req. Rate** (1/s)	**Conn. Rate** (1/s)	**Resp. Rate** (1/s)	**Resp. Time** (ms)	**IO** (KB/s)	**Error Rate** (%)
60	60.0	20.0	60.0	5.3	33.0	0.00
90	90.0	30.0	90.0	5.2	49.5	0.00
120	120.0	40.0	120.0	5.3	66.0	0.00
150	150.0	50.0	150.0	7.2	82.5	0.00
180	180.0	60.0	179.9	8.3	98.9	0.00
210	209.9	70.0	209.9	8.6	115.4	0.00
240	239.8	80.0	239.5	13.3	131.8	0.02
270	269.6	89.9	269.6	14.9	148.2	0.00
300	271.2	92.5	270.0	397.7	149.0	0.82
330	261.7	98.1	258.4	523.6	142.3	4.64
360	257.2	104.1	252.2	558.5	138.2	8.25

Table A.1.: Results of performance tests requesting the "index.html" file.

Demanded Req. Rate (1/s)	**Req. Rate** (1/s)	**Conn. Rate** (1/s)	**Resp. Rate** (1/s)	**Resp. Time** (ms)	**IO** (KB/s)	**Error Rate** (%)
60	60.0	20.0	60.0	5.5	618.6	0.00
90	90.0	30.0	90.0	5.7	927.8	0.00
120	120.0	40.0	119.9	5.9	1236.6	0.00
150	150.0	50.0	150.0	8.3	1545.9	0.00
180	179.9	60.0	179.9	15.1	1854.0	0.00
210	192.1	65.6	191.4	528.2	1975.0	1.00
240	189.7	72.8	186.8	650.9	1922.6	6.00
270	187.8	80.6	180.3	676.4	1865.9	11.00
300	187.0	86.9	178.5	697.1	1843.0	15.00
330	185.0	93.6	173.5	710.4	1781.2	21.00
360	185.0	100.5	170.0	724.2	1754.9	26.00

Table A.2.: Results of performance tests requesting the "10K.html" file.

Demanded Req. Rate (1/s)	**Req. Rate** (1/s)	**Conn. Rate** (1/s)	**Resp. Rate** (1/s)	**Resp. Time** (ms)	**IO** (KB/s)	**Error Rate** (%)
60	60.0	20.0	60.0	5.3	23.3	0.00
90	90.0	30.0	90.0	5.4	35.0	0.00
120	120.0	40.0	120.0	5.3	46.6	0.00
150	150.0	50.0	150.0	6.2	58.3	0.00
180	180.0	60.0	179.9	7.2	69.9	0.00
210	209.9	70.0	209.9	8.3	81.6	0.00
240	239.8	80.0	239.8	12.1	93.2	0.01
270	269.3	89.8	269.8	11.7	104.7	0.00
300	296.7	98.9	296.3	76.5	115.3	0.00
330	295.7	100.2	297.9	380.8	114.8	0.61
360	301.9	107.3	298.9	442.4	116.7	2.47

Table A.3.: Results of performance tests requesting a non-existent file.

Results of dedicated throughput tests

Conn.	**usr** (%)	**sys** (%)	**nic** (%)	**idle** (%)	**io** (%)	**irq** (%)	**sirq** (%)	**Total** (%)
1	2.9	8.5	0	63.8	0	0	24.7	36.1
2	3	25	0	16.6	0	0	55.2	83.2
3	1.9	40.3	0	0.9	0	0	56.7	98.9
4	1.3	40.9	0	0.1	0	0	57.4	99.6
5	2.9	40.1	0	0	0	0	56.8	99.8
6	2.5	43.4	0	0	0	0	54	99.9
7	1.9	42.3	0	0	0	0	55.7	99.9
8	1.7	41.7	0	0	0	0	56.4	99.8
9	1.5	37.7	0	0	0	0	60.7	99.9
10	1.7	40	0	0	0	0	58.1	99.8

Table A.4.: Results (CPU utilization) of dedicated network throughput tests.

usr: user space, **sys**: system/kernel, **nic**: low priority (nice), **idle**: idle time, **io**: waiting for i/o, **irq**: serving irqs, **sirq**: serving soft irqs

Conn.	**nginx** (Mbps)	**IIS** (Mbps)
1	24.451	43.730
2	48.424	162.263
3	56.835	329.177
4	60.965	413.588
5	57.501	490.823
6	57.809	476.802
7	61.128	469.405
8	61.286	511.762
9	57.928	502.813
10	57.652	515.770

Table A.5.: Results (Mbps) of dedicated network throughput tests.

B Resources

B.1 Supplied Digital Assets

Provided with this thesis is a CD containing all digital assets mentioned in the previous chapters or necessary for replicating the accomplished results. Besides the delivery on CD, required by formalities of delivering a thesis, everything is available for download, too.

Data on the CD is organized in the following structure:

B.1.1 Hardware

⇒ Download at `http://www.peschuster.de/ies-thesis/Hardware.zip` (20 KB)

The `Hardware` directory contains project files for *Xilinx ISE Project Navigator* and *Xilinx Platform Studio* in version 14.1, as well as all required configuration files (mhs and ucf).

Additionally a `xmd.ini` is included, which can be placed in a directory and automates programming the *MicroBlaze* processor with the custom Linux image file (note: the contained path must be adjusted to the own environment).

B.1.2 Software

⇒ Download at `http://www.peschuster.de/ies-thesis/Software.zip` (917 KB)

Linux

The `Linux` sub-directory contains

- The complete Linux kernel `.config` file (`linux.config`)
- The *Device Tree Source* file (`xupv5.dts`)
- A script for packing a directory structure in a gzipped *cpio* archive (`pack-fs.sh`).
- A TCL helper function for usage with *XMD* to dump the Linux system variable `__log_buf` from memory into a local file and decode its content (`syslog.tcl`).
- A patch enabling processor version recognition for recent *MicroBlaze* processor versions (`new-microblaze-versions.patch`).
- The published patch by *Chelsio Communications* integrating their *TOE* solution into the Linux kernel (`toe-chelsio.patch`).

nginx

The `nginx` sub-directory contains

- A snapshot of the nginx source code repository containing the latest version with all integrated changes (`source/`).
- The patch for incorporating a "memory guard" in a separate file (`0001-Added-a-memory-guard.patch`).

- A small C program to be executed on the *MicroBlaze* processor evaluating correct values for configuration of the nginx build (mb-env-analysis.c).
- Debug log of nginx for startup, two subsequent requests on one connection and shutdown as an Excel file incl. color highlighting for some message categories and process ids (nginx-debug-log.xlsx).

B.1.3 Images

⇒ Download at http://www.peschuster.de/ies-thesis/Images.zip (41,987 KB)

The Images directory contains

- The *bitstream* files for the *Xilinx XUPV5-LX110T* board (download.bit).
- The Linux kernel image file with working nginx installation and static IP address 192.168.2.125 (simpleImage.xupv5).
- The root file system as linked into the Linux kernel image file in a separate, gzipped *cpio* archive (complete_fs.cpio.gz).

B.1.4 Performance-Tests

⇒ Download at http://www.peschuster.de/ies-thesis/Performance-Tests.zip (41 KB)

The Performance-Tests directory contains

- The C# console application generating requests for high load on the nginx system as source code and binary (CsharpLoadGenerator/).
- The test HTML file having a file size of exactly 10.240 bytes (10K.html)
- Scripts with automated performance tests using *httperf* (*.sh)
- Results of the performance tests in form of log files (results/*.log), summarized key figures (results/*.tsv) and graphs (results/*.ps).

B.2 Auxiliary Scripts

The following section contains the source code of two of the supplied scripts for easier access:

B.2.1 SoC Environment Evaluation Program

C program

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include "xparameters.h"
#include "xil_cache.h"

void print(char *str);

int main()
{
    #ifdef XPAR_MICROBLAZE_USE_ICACHE
```

```
        Xil_ICacheEnable();
    #endif
    #ifdef XPAR_MICROBLAZE_USE_DCACHE
        Xil_DCacheEnable();
    #endif

    print("Env analysis (size_of): \r\n \r\n");

    printf("size of int: %d \r\n", (int)sizeof(int));
    printf("size of long: %d \r\n", (int)sizeof(long));
    printf("size of long long: %d \r\n", (int)sizeof(long long));
    printf("size of void *: %d \r\n", (int)sizeof(void *));
    printf("size of sig_atomic_t: %d \r\n", (int)sizeof(sig_atomic_t));
    printf("size of size_t: %d \r\n", (int)sizeof(size_t));
    printf("size of off_t: %d \r\n", (int)sizeof(off_t));
    printf("size of time_t: %d \r\n", (int)sizeof(time_t));

    Xil_DCacheDisable();
    Xil_ICacheDisable();

    return 0;
}
```

Result

```
size of int: 4
size of long: 4
size of long long: 8
size of void *: 4
size of sig_atomic_t: 4
size of size_t: 4
size of off_t: 4
size of time_t: 4
```

B.2.2 C# Load Test Console Application

```
using System;
using System.Net;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        while (true)
        {
            Console.WriteLine("Specify params. Form: [number]:[threads]");
            string input = Console.ReadLine();

            if (string.IsNullOrWhiteSpace(input))
                return;
            string[] p = input.Split(':');
            int req = int.Parse(p[0]);
            if (req <= 0)
                return;

            long count = 0;
            DateTime start = DateTime.Now;

            int parallel = int.Parse(p[1]);
            string baseUrl = "http://192.168.2.125/10K.html?id=";
            var parellelOptions = new ParallelOptions { MaxDegreeOfParallelism = parallel };

            Parallel.For(0, req, parellelOptions, i =>
            {
                var http = HttpWebRequest.CreateHttp(baseUrl + i);

                using (http.GetResponse())
                {
                }

                double c = Interlocked.Increment(ref count);

                if (i % 10 == 0)
                {
                    Console.WriteLine(
                        "{0, 8} {1:0.000}",
                        c,
                        c / (DateTime.Now - start).TotalSeconds);
                }
            });
        }
    }
}
```